M T S

The Michigan Terminal System

Volume 6: FORTRAN in MTS

October 1983

Updated February 1988 (Update 1)

The University of Michigan Computing Center
Ann Arbor, Michigan

****
*
* This obsoletes the December 1978 edition. *
*
****

DISCLAIMER

The MTS Manual is intended to represent the current state of the Michigan Terminal System (MTS), but because the system is constantly being developed, extended, and refined, sections of this volume will become obsolete. The user should refer to the U-M Computing News, Computing Center Memos, and future Updates to this volume for the latest information about changes to MTS.

PREFACE

The software developed by the Computing Center staff for the operation of the high-speed processor computer can be described as a multiprogramming supervisor that handles a number of resident, reentrant programs. Among them is a large subsystem, called MTS (Michigan Terminal System), for command interpretation, execution control, file management, and accounting maintenance. Most users interact with the computer's resources through MTS.

The MTS Manual is a series of volumes that describe in detail the facilities provided by the Michigan Terminal System. Administrative policies of the Computing Center and the physical facilities provided described in other publications.

The MTS volumes now in print are listed below. The date indicates the most recent edition of each volume; however, since volumes are updated by means of CCMemos, users should check the file *CCPUBLICATIONS, or watch for announcements in the U-M Computing News, to ensure that their MTS volumes are fully up to date.

Volume 1: The Michigan Terminal System, January 1984
Volume 2: Public File Descriptions, January 1987
Volume 3: System Subroutine Descriptions, April 1981
Volume 4: Terminals and Networks in MTS, March 1984
Volume 5: System Services, May 1983
Volume 6: FORTRAN in MTS, October 1983
Volume 7: PL/I in MTS, September 1982
Volume 8: LISP and SLIP in MTS, June 1976
Volume 9: SNOBOL4 in MTS, September 1975
Volume 10: BASIC in MTS, December 1980
Volume 11: Plot Description System, August 1978
Volume 12: PIL/2 in MTS, December 1974
Volume 13: The Symbolic Debugging System, September 1985
Volume 14: 360/370 Assemblers in MTS, May 1983
Volume 15: FORMAT and TEXT360, April 1977
Volume 16: ALGOL W in MTS, September 1980
Volume 17: Integrated Graphics System, December 1980
Volume 18: The MTS File Editor, February 1988
Volume 19: Tapes and Floppy Disks, November 1986
Volume 20: Pascal in MTS, December 1985
Volume 21: MTS Command Extensions and Macros, April 1986
Volume 22: Utilisp in MTS, February 1988
Volume 23: Messaging and Conferencing in MTS, March 1987

Other volumes are in preparation. The numerical order of the volumes does not necessarily reflect the chronological order of their

appearance; however, in general, the higher the number, the more specialized the volume. Volume 1, for example, introduces the user to MTS and describes in general the MTS operating system, while Volume 10 deals exclusively with BASIC.

The attempt to make each volume complete in itself and reasonably independent of others in the series naturally results in a certain amount of repetition. Public file descriptions, for example, may appear in more than one volume. However, this arrangement permits the user to buy only those volumes that serve his or her immediate needs.

Richard A. Salisbury

General Editor

PREFACE TO VOLUME 6

Volume 6 contains the descriptions of the various system components related to the use of the FORTRAN programming language in MTS.

Acknowledgments for the descriptions and programs contained in this volume are as follows:

The section "Interactive FORTRAN" is taken from the document UBC IF by Dennis O'Reilly (June 1975) which was produced by the University of British Columbia Computing Centre. The program was developed by the programming staff at that installation.

The subsections "FORTRAN-G Source Module Error/Warning Messages," "FORTRAN-H Optimization Facilities," and "FORTRAN-H Source Module Error/Warning Messages" are reprinted with permission from the IBM publication, IBM System/360 Operating System FORTRAN IV (G and H) Programmer's Guide, form GC35-0002.

The subsection "The FORTRAN Debug Facility" is reprinted with permission from the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

| The subsection "Compiler Options" of section "VS FORTRAN" is
| reprinted with permission from the IBM publication, VS FORTRAN
| Version 2: Programming Guide, form SC26-4222.

The remainder of the descriptions in this volume were either produced or extensively modified from other documentation by the editorial and programming staffs at the University of Michigan Computing Center.

Page Revised February 1988 October 1983

Contents

OVERVIEW OF FORTRAN IN MTS

This volume is intended for those users who wish to use the FORTRAN IV language in MTS. It is assumed that the user is already familiar with the FORTRAN language; therefore, only those cases where the language accepted by a particular FORTRAN language processor differs from the standard language specifications are described in detail.

Documentation for the FORTRAN language is available from several sources. The following reference publications are available through the local bookstores.

IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

This IBM publication describes the FORTRAN IV language which is accepted by the IBM FORTRAN-G and FORTRAN-H compilers. This is the language reference publication.

| VS FORTRAN Version 2: Language and Library Reference, form SC26-4221.

This IBM publication describes the FORTRAN 77 language which is accepted by the IBM VS FORTRAN compiler.

| FORTRAN 77 With MTS and the IBM PC, Brice Carnahan and James O. Wilkes (1985).
|
| This publication provides an introduction to the FORTRAN 77 language and MTS.

FORTRAN IV with WATFOR and WATFIV, Paul Cress, Paul Dirksen, and J. Wesley Graham, Prentice-Hall (1970).

This publication provides an introduction to programming with WATFIV.

There are five different FORTRAN language processors available in MTS. These are FORTRAN-G, FORTRAN-H, VS FORTRAN, WATFIV, and IF. Each of these is oriented toward the differing needs of the FORTRAN user. A brief description of each is given below.

(1) FORTRAN-G

FORTRAN-G is the IBM standard FORTRAN IV compiler and the one most often used for production FORTRAN programs. The compiler produces relatively efficient object code. Object modules produced may be debugged interactively via the Symbolic Debug-

ging System (SDS). This compiler is generally invoked via the *FTN compiler interface program.

(2) FORTRAN-H

FORTRAN-H is the IBM optimizing FORTRAN IV compiler. This compiler is intended for programs which are fully debugged and ready for production use. The compiler produces very efficient object code which generally executes 25% to 50% faster than the equivalent FORTRAN-G object code. Compilation with FORTRAN-H is more expensive than with FORTRAN-G due to the complicated optimizing operations performed by the compiler. The difference in cost is the greatest for programs which consist of a single large main program and the least for programs which are subdivided into smaller subroutines. Object modules produced may be debugged interactively via the Symbolic Debugging System, although not as easily as with FORTRAN-G object modules. This compiler is generally invoked via the *FTN compiler interface program.

(3) VS FORTRAN

The IBM VS FORTRAN compiler supports the most recent standard for FORTRAN 77 published by the American National Standards Institute (ANSI). It also supports IBM extensions to the language and contains features and extensions that are not available with FORTRAN 66 compilers (FORTRAN-G and FORTRAN-H).

(4) WATFIV

The WATFIV compiler (Waterloo FORTRAN IV) is a compiler oriented toward batch program development. It provides fast compilation, stringent error checking, and good diagnostics. Since WATFIV produces object code which incorporates extensive error checking, it is not economical for fully debugged production programs. WATFIV programs generally must be self-contained and cannot easily call subroutines produced by other language processors. The language accepted by WATFIV differs somewhat from that accepted by other FORTRAN compilers. The extensions and restrictions are described in the section "WATFIV."

(5) IF

The IF (Interactive FORTRAN) compiler is a processor oriented toward interactive program development. IF enables the user to enter entire programs either from a terminal or a file, to dynamically debug and correct the errors, and to save the debugged source program. IF is an interpretive processor; it will not produce object modules and does not execute the compiled program efficiently. However, it is very flexible and useful for error-checking purposes. There are two versions of this processor. *IF66 accepts the FORTRAN 66 (FORTRAN-IV) standard which is the same as the source language for the

| FORTRAN-G and FORTRAN-H compilers. *IF77 accepts the FORTRAN 77
| standard which is the same as the source language for VS
| FORTRAN.

In addition to the descriptions of the above compilers, this volume also contains descriptions of the *FTN compiler-interface program, the MTS FORTRAN I/O library, the OVERDRIVE preprocessor program, and several useful subroutines for FORTRAN programs.

The *FTN compiler interface program enables the user to invoke either the FORTRAN-G or FORTRAN-H compilers. It also provides several useful facilities for both conversational and batch users which are not provided directly by the compilers themselves. The compiler interface is described in the section "*FTN Interface."

The FORTRAN I/O library provides an interface to the MTS I/O facilities for FORTRAN programs and provides a limited amount of error recovery from I/O errors for conversational users. This is described in the section "FORTRAN I/O Library."

The OVERDRIVE preprocessor is a program that allows the use of structured programming techniques in FORTRAN. This is described in the section "OVERDRIVE."

Page Revised February 1988 October 1983

October 1983

*FTN INTERFACE

INTRODUCTION

The FORTRAN compiler interface program in *FTN is a single program which allows the user to employ either of the two IBM FORTRAN compilers--FORTRAN G and FORTRAN H--available in MTS. This interface is designed to give the user a standard means of interaction with both compilers. The interface also provides a large range of user services. Among the auxiliary services provided are:

(1) conversational entry and correction of interface and compiler options,

(2) four source statement formats,

(3) availability of the reformatted source program in standard IBM format,

(4) incorporation of the MTS line numbers in the source listings and terminal diagnostics,

(5) an object module format employing CSI-type (core storage image) loader records to decrease loading time,

(6) availability of SYM (symbol) records in the object module, and

(7) availability of different optimization levels from the FORTRAN-H compiler.

At a few points, the reader is referred to the sections "FORTRAN G" and "FORTRAN H" in this volume. These are references to detailed descriptions of some aspects of the compiler output listings, and thus may be ignored unless such detailed knowledge is desired. In no case are they crucial to an understanding of the features of *FTN considered herein. The explanations of the error messages in these sections may be useful in identifying problems noted by the compiler diagnostic messages.

For a complete description of the FORTRAN programming language, see the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form number GC28-6515.

DESCRIPTION OF THE INTERFACE

The *FTN interface is invoked by a command of the form

```
$RUN *FTN SCARDS=source SPRINT=listing SPUNCH=object PAR=options
```

The logical I/O units are used primarily for specifying the location of the source program, the destination for compiler output listings and diagnostics, and the destination for the generated object module. The options are used primarily for specifying the compiler to be used and the compiler services that are desired.

The logical I/O unit assignments are as follows:

- SCARDS - source program (defaults to *SOURCE*)
- SPRINT - compiler source program and object module listings (defaults to *SINK* in batch mode)
- SPUNCH - object module generated (defaults to *PUNCH* in batch mode)

In addition, four other logical I/O units may be assigned. They are as follows:

- SERCOM - interface and compiler diagnostic error messages (defaults to *MSINK*)
- GUSER - responses to interface prompting messages (defaults to *MSOURCE*)
- 0 - object module generated (if DECK option is specified)
- 1 - edited source module generated (if EDIT option is specified)

The logical I/O units GUSER and SERCOM are used to communicate with the conversational user. When an error message or request message is printed on SERCOM, *FTN will prompt the user for an appropriate reply via GUSER. This method of interface-user communication is used for option entry and correction. GUSER and SERCOM default to *MSOURCE* and *MSINK*, respectively; however, they may be reassigned to other files or devices.

When *FTN requests the user to enter options, it reads from GUSER using the characters ":PAR=" to prompt. The response may be continued on successive lines by using the current MTS command line continuation character (default is "-"); however, the total length of the response must not exceed 240 characters. If the response is an MTS command, i.e., if it begins with a dollar sign "$", then it is executed as such and control is returned to MTS command mode. Control similarly returns to MTS with an end-of-file response. In either case, if execution is resumed via a $RESTART command, another GUSER prompt will be made.

In scanning the response to a GUSER prompt, *FTN assembles all unknown option names and any keyword option names assigned illegal values into a single character string. This string is subsequently

enclosed in quotation marks, suffixed with a question mark, and printed on SERCOM. In addition, any problems associated with the assignment or defaulting of any of the input/output options are noted on SERCOM. If the option scan detects any errors, the conversational user will be given an opportunity to correct them, i.e., another GUSER prompt will be made.

If the logical I/O unit SCARDS is assigned to the terminal, *FTN uses the question mark "?" to prompt for the next source statement line. The null response is treated as an end-of-file and thus serves to terminate the source stream. If a command is entered, i.e., if the response begins with a dollar sign, the current source module is regarded as complete. When compilation is completed, the command is executed and control passes to MTS. The current MTS command line continuation character may be used for these commands; however, successive lines of the command are not read until the compilation has been completed. The total length of the command must not exceed 240 characters. Note that in batch mode, the QUIT option may be used to prevent the execution of any command given.

The logical I/O unit SERCOM is also used for various comments and diagnostics generated during the compilation. By default, conversational users will also have all source program diagnostics reproduced on SERCOM in a condensed form, which, for the LINE and EDITED statement formats (see the section "Source Statement Formats"), includes the MTS line numbers. This does not occur in batch mode since SERCOM defaults to *MSINK*.

Attention interrupts are processed by *FTN only while it is printing EXPLAIN output on SERCOM. If an attention is received, the printing is stopped and a return is made to MTS command mode. A subsequent $RESTART will result in a prompt for more options. In all other cases, MTS processes attention interrupts.

OPTIONS

The compiler options provide for user control of the optional compiler services.

Option names which are not recognized by *FTN or which are assigned illegal values are noted on SERCOM. Conversational users will be subsequently prompted to enter new or additional options to correct their errors. Any unrecognizable option values or illegal assignments result in termination of batch mode jobs unless the NOQUIT option (see the section "Simple Option Descriptions" below) has been specified, in which case the erroneous options and any others dependent upon them are ignored.

Simple options may be negated by prefixing "NO", "¬", or "-". The option names may be specified in any order; however, if the same option

is repeated, the previous occurrence(s) in the left to right scan are overridden. The options must be separated either by blanks or commas, and blanks should not be embedded in a option name or anywhere within a option assignment. The option names may be abbreviated by truncation from the right. The following table gives the minimum acceptable abbreviations and the defaults used if the option is not specified.

Simple Option	Shortest Abbreviation	Batch Default	Conversational Default
BCD	B	NOBCD	NOBCD
COMMENT (H only)	COM	COMMENT	COMMENT
COND (G only)	C	COND	COND
DECK	D	NODECK	NODECK
EDIT	E	NOEDIT	NOEDIT
EJECT (H only)	EJ	EJECT	EJECT
ERR	ER	ERR	ERR
EXPLAIN	EX or ?	NOEXPLAIN	NOEXPLAIN
ID	I	ID	ID
LIB	LIB	NOLIB	NOLIB
LIST	L	NOLIST	NOLIST
LOAD	LO	LOAD	LOAD
MAP	M	NOMAP	NOMAP
MTS	MT	NOMTS	NOMTS
OVER	OV	NOOVER	NOOVER
QUIT	Q	QUIT	NOQUIT
SCAN	SC	NOSCAN	NOSCAN
SM	SM	NOSM	NOSM
SML	SML	NOSML	NOSML
SOURCE	S	SOURCE	NOSOURCE
STRUC (H only)	ST	NOSTRUC	NOSTRUC
TEST	T	NOTEST	NOTEST
XL (H only)	XL	NOXL	NOXL
XREF (H only)	X	XREF	NOXREF

Assignment Option	Shortest Abbreviation	Batch Default	Conversational Default
CALIGN (H only)	CA	0	0
CSHIFT (H only)	CS	0	0
FORMAT	F	EDITED	EDITED
LINE	LI	57	57
NAME	N	MAIN	MAIN
OPT	O	G	G
OVER	OV	(See text)	(See text)
SIZE (G only)	SI	4	4

Simple Option Descriptions

Options which are employed by default in both batch and conversational use are noted as such.

BCD

The BCD option indicates that the source module lines are coded in Binary Coded Decimal (as on the IBM 026 keypunch). The standard for the System/370 is Extended Binary Coded Decimal Interchange Code (EBCDIC). Most terminals and the 029 keypunch produce EBCDIC code. If the BCD option is specified, statement numbers passed as arguments must be coded as "$n", and the dollar sign "$" must not be used as an alphabetic character. With EBCDIC, statement numbers in argument lists would be coded "&n", so that "$" would be a legitimate alphabetic character. The default is NOBCD.

The FORTRAN-G and FORTRAN-H compilers do not support BCD characters in either literal data or as print control characters; such characters are treated as EBCDIC. Consequently, for example, a BCD "+", used as a carriage-control character will not cause printing to continue on the same line. Programs keypunched in BCD should be carefully scanned for possible errors relating to print control characters and literal data.

COMMENT

The NOCOMMENT option inhibits the listing of comment statements when the FORTRAN-H compiler is used. The default is COMMENT.

COND

The COND option specifies that compilation is to be terminated without producing an object module if serious errors (those with severity levels of 4 or 8) are found in the source program. A list of the FORTRAN-G diagnostic messages and severity levels is given in the section "FORTRAN G." The default is COND. The option applies only to FORTRAN-G.

DECK

If the DECK option is specified, each object module is suffixed with the first four characters of the module name in columns 73-76 and the line sequence number in columns 77-80. DECK has no effect if the SCAN option is enabled. The default is NODECK.

EDIT

The EDIT option indicates that the edited source modules are to be written on logical I/O unit 1. The default is NOEDIT.

EJECT

The NOEJECT option causes the FORTRAN-H compiler to change all page ejects to triple spaces. This may be useful for some terminals. The default is EJECT.

ERR

The ERR option requests that source diagnostics are to be printed on SERCOM. It is assumed that this logical I/O unit corresponds to *SINK* so that if the output listings are also assigned to *SINK*, the ERR option will be ignored. The default is ERR.

EXPLAIN

The EXPLAIN option prints a synopsis of the interface and compiler options on SERCOM. An attention interrupt can be used to stop the printing of this information. After the explanation has been printed, the interface again prompts the conversational user for options. The default is NOEXPLAIN.

ID

The ID option generates internal statement numbers (ISN) following external function references in the object code produced by the compiler. (An internal statement number is that number which would be attached to the statement if each executable statement were numbered sequentially from the beginning of the program.) In the generated code, each BALR (branch) to an external program is followed by a four-byte no-operation with an address field equal to the ISN of the source statement containing the external reference. For example, for a call from statement 1000, the four bytes would appear in a hexadecimal dump as 470003E8 (3E8 of base 16 = 1000 of base 10). This is useful as a debugging aid, but it is not necessary when debugging with SDS. The default is ID.

LIB

The LIB option (available only for FORTRAN-G, OPT=G, the default) specifies that each object module generated by the compiler is to be preceded by a LIB record containing its module name. This option affects both the LOAD and DECK output. The LIB record immediately precedes the first ESD record of the object module. The format of the LIB record is as follows:

Columns	Contents
2-4	LIB
17-24	the module name (1 to 8 characters).

For main programs the module name is determined by the value of the NAME assignment option. For subroutines it is always the name

given in the FUNCTION or SUBROUTINE statement. The default is NOLIB.

LIST

The LIST option includes a pseudo-assembly-language format listing of the generated object module in the output listings. Unless the user understands some machine code and can read hexadecimal dumps, this listing is generally not useful and is in all cases expensive to obtain. The use of the SCAN option overrides the LIST option as there is no object module produced when that option is employed. The COND option can prevent production of the object module and hence the listing. The default is NOLIST.

LOAD

The LOAD option specifies that object modules are to be produced by the compiler and included in the object module output. LOAD has no effect if used with the SCAN option. If the DECK option is specified, each record will be 80 characters long; otherwise, the object records will be variable in length with a possible maximum length equal to the maximum record length of the file or device receiving the object module. The default is LOAD.

Object module lines contain a 12-2-9 punch in the first column and the characters ESD, TXT, RLD, or END, in columns 2-4. The compilers generate four types of ESD items: type 0 items contain the module name, entry point, and module length; type 1 items contain the entry point names corresponding to ENTRY statements; type 2 items contain the external references made in CALL or EXTERNAL statements and implicit or explicit function references; and type 5 items contain the names for each COMMON block. The compilers do not generate type 3 or type 4 ESD items. The TXT records contain user- and compiler-generated constants, translated FORMAT statements, and the generated machine instructions. The information contained in the RLD records is used by the loader to complete external references. External references are resolved by adjusting the constant pointed to by the address in the RLD item by the address of the appropriate external symbol contained in one of the type 2 ESD items. The END record for each module is described below:

Columns	Contents
1	12-2-9
2-4	END
37-39	FTN
41-48	Module name
49-56	Date as MM-DD-YY
57-64	Time as HH:MM.SS
65-68	Number of warning errors
69-72	Number of serious errors

If the output record length is less than 255, the object is generated in the form of 80-byte card images. Otherwise, the interface automatically concatenates successive RLD records to form 255-character RLD records and successive TXT records to form either 255-character TXT records or, if the object text is longer, CSI records. (See MTS Volume 5, System Services, section entitled "The Dynamic Loader," for an explanation of the different types of loader records.) This editing is applied to all object modules included in the LOAD data set.

Any LIB records generated will contain the characters LIB in columns 2-4 and the module name in columns 17-24.

Note that when the SCAN option is specified, no punched output will be generated regardless of the LOAD, DECK, or LIB option specifications. Similarly, if the COND option is specified, then no punched output will be generated for any source module containing serious errors, i.e., diagnostics with severity level 4 or 8.

MAP

The MAP option specifies that a storage map is to be included in the output listings. The map consists of tables containing variable names and locations for COMMON, EQUIVALENCE, NAMELIST, scalar and array variables, subprograms referenced, and FORMAT statements. The default is NOMAP.

MTS

The MTS option returns control immediately to MTS. If the *FTN interface is restarted by the $RESTART command, the user is again prompted for a option list which will augment all previously entered options. The default is NOMTS.

OVER

The OVER option causes *FTN to invoke the OVERDRIVE preprocessor on the source program before it passes it to the appropriate compiler. The default is NOOVER. See the section "OVERDRIVE" in this volume for the description of the OVERDRIVE preprocessor.

QUIT

The QUIT option terminates a batch job if serious errors, i.e., severity levels 4 and 8, are found in the source program. In such a case the entire job is terminated, no more commands are executed, and the user is signed off. In addition, if option errors are found in batch mode, the job is terminated without invoking the compiler. For example, the batch command

```
$RUN *FTN SPUNCH=MYFILE PAR=QUIT
```

causes termination if the user file MYFILE does not exist. If NOQUIT is specified and option errors are found, then compilation continues if the source data set is available. However, any output options for which data sets are not readily available and any unrecognized options are ignored. In the example given above, no LOAD output would be generated. The default is QUIT for batch mode and NOQUIT for conversational mode.

SCAN

The SCAN option causes the compiler to scan the source module for syntax and compilation errors. All appropriate error diagnostics are generated for each source module, but no object code is generated and consequently neither the object listing nor the program size can be included in the output listings. In addition, no object module output is produced. Use of the SCAN option will decrease the cost of scanning the source program for errors. The default is NOSCAN.

SM

The SM option is an abbreviation for SOURCE and MAP combined.

SML

The SML option is an abbreviation for SOURCE, MAP, and LIST combined.

SOURCE

The SOURCE option specifies that a source listing is to be included in the output listings. The source listing always corresponds to the edited source module. When the source format is either LINE or EDITED (see the section "Source Statement Formats"), the MTS line numbers are incorporated in this listing immediately to the right of the source statement. The first line number given corresponds to the line containing the first character appearing in the reformatted source statement, while the second, if given, corresponds to the line from which the last character was obtained. If any LIST output is to be generated, then all source diagnostics are included. The default is SOURCE for batch mode and NOSOURCE for conversational mode.

STRUC

The STRUC option is effective only with optimization level 2 of the FORTRAN-H compiler (see the description of the OPT assignment option). It requests a structured source listing which is included on the PRINT data set. The default is NOSTRUC.

TEST

The TEST option generates SYM records in the object module. This facilitates execution-time debugging with the Symbolic Debugging System (SDS). See the section "Introduction to Debug Mode for FORTRAN" for further information. The default is NOTEST.

XL

The XL option is used only by the FORTRAN-H compiler and allows the use of extended language features which are supported by FORTRAN-H and described in IBM System/360 Operating System FORTRAN IV (H) Compiler Program Logic Manual, form GY28-6642. This is not the same as the FORTRAN-H Extended compiler (an IBM program product), which is not available at the Computing Center. The default is NOXL.

XREF

The XREF option is used only by the FORTRAN-H compiler and generates a cross-reference listing of variable names and statement numbers. The cross-reference listing appears on the PRINT data set. This option has no effect when used with the FORTRAN-G compiler. See the description of the OPT keyword option for an explanation of the method for specifying which compiler is to be used. The default is XREF for batch mode and NOXREF for conversational mode.

Assignment Option Descriptions

CALIGN=n
CSHIFT=n

By specifying the CALIGN=n or CSHIFT=n options (where $1 \leq n \leq 72$), the body of comment lines may be positioned to start in column "n". The comment lines are denoted by column 1 containing the comment symbol "C". CALIGN aligns the first nonblank character after the initial "C" at column "n". CSHIFT shifts the entire comment (including the "C") to column "n", thus preserving the indentation of structured comments. For both CALIGN and CSHIFT, a "C" is printed at column 1 in the listing. Blank comments are not shifted. If "n" is 72, the comments will be printed in the normally blank area to the right of the source statement listing. If truncation would occur, the comment is continued on the next line. Standard output is printed with CALIGN=0 and CSHIFT=0 (the defaults). CALIGN=1 is treated as if CALIGN=2 were specified. If "n" is greater than 72, 72 will be used. CALIGN and CSHIFT are "coupled" options, i.e., the last one specified overrides all previous occurrences of either. These options are available only with the FORTRAN-H compiler.

FORMAT={LINE|IBM|EDITED|LONG}

The value assigned to this option specifies which of the four source statement formats (LINE, IBM, EDITED, and LONG) should be used; statement formats are fully described in the section "Source Statement Formats." The default is EDITED.

LINE=n

The LINE assignment option specifies the number of lines per page for output listings. "n" must be an integer in the range (3,32767). This may be used to control the spacing of the page headers, etc., as they are only produced at the beginning of each set of lines. The default is 57 lines per page.

NAME=xxxx

The NAME assignment option specifies the name to be used as the module name for all main programs compiled. Subprogram module names are always the name given in the FUNCTION or SUBROUTINE statement. The module name appears in the page headers, LIB records, the ESD type 0 lines, the END lines, and the object code at relative address 000005. The name "xxxx" may be from 1 to 8 characters in length. The default is MAIN.

OPT={G|0|1|2|H}

The OPT assignment option specifies either the FORTRAN-G compiler (OPT=G) or one of the three levels of optimization available with the FORTRAN-H compiler (OPT=0, OPT=1, or OPT=2). Optimization level 0 provides the least optimization; level 2 provides the most optimization. For convenience, OPT=H may be used interchangeably with OPT=2, since level 2 is used by most FORTRAN-H users. The default is G.

OVER=parlist

The OVER assignment option may be used to pass options to the OVERDRIVE preprocessor. A single option is placed directly after the "=" character; multiple options must be enclosed in parentheses, e.g.,

```
OVER=LIST
OVER=(LIST,COM)
```

All options passed to OVERDRIVE must be included in a single assignment. *FTN automatically passes the appropriate COMPILER=FTNG or COMPILER=FTNH option to OVERDRIVE. By default, other no options are passed. See the section "OVERDRIVE" in this volume for the description of the OVERDRIVE preprocessor.

SIZE=n

The SIZE assignment option specifies the number of pages of virtual memory to be used for the FORTRAN-G working storage. "n" must be an integer in the range (1,255). The default is 4 pages. This option is more fully described in the section "FORTRAN G." It is not used by the FORTRAN-H compiler.

INPUT/OUTPUT ASSIGNMENT

The *FTN interface handles from one to five data sets, depending on user-supplied options. These data sets may be explicitly assigned via the logical I/O units or may be defaulted as follows:

	Batch	Conversational
SCARDS	*SOURCE*	*SOURCE*
SPRINT	*SINK*	-PRINT
SPUNCH	-LOAD	-LOAD
0	*PUNCH*	-DECK
1	*PUNCH*	-EDIT

Notice that these defaults are not the same as the standard MTS defaults for these logical I/O units. For example, if the conversational user did not assign SPRINT to an FDname but did request compiler options specifying printed output (e.g., SOURCE, MAP, or LIST), the file -PRINT is created and used. If -PRINT already exists, it is emptied prior to use. If no compiler options specifying printed output were requested, the printed output is suppressed. For both the batch user and the conversational user, SPUNCH defaults to the file -LOAD. If -LOAD already exists, it is emptied before compilation begins. If the user explicitly specifies the files to be used for the printed output or the object module output, the files are not emptied by the interface; it is the user's responsibility to do this.

In general, the LOAD data set corresponds to SPUNCH and the DECK data set corresponds to logical I/O unit 0. This correspondence may be reversed if the user specifies the DECK option and does not assign logical I/O unit 0. In this case, if LOAD has not been specified, the DECK data set is generated on SPUNCH.

A data set, if not assigned to an FDname, defaults to a file or device depending on the type of job (batch or conversational). Data set defaults are accomplished by setting the corresponding logical I/O unit to use the default file/device. Unassigned output data sets which default to temporary files are emptied automatically before use.

SOURCE STATEMENT FORMATS

Four types of source statements are supported by *FTN: IBM, LONG, LINE, and EDITED. These are specified by the FORMAT assignment option; the default is EDITED.

The IBM format corresponds to the card-oriented format in general use and passes statements directly to the compiler. The LINE format allows free placement of the optional statement number and statements within the line, and a continuation convention similar to the standard system technique for MTS command lines. The EDITED format is an attempt to bridge the gap between LINE and IBM formats by using the format which appears to be appropriate for the source line being processed. A series of tests is used to determine which format is to be employed. LONG format allows IBM formatted lines to extend beyond column 72.

Because *FTN employs IBM FORTRAN compilers (which accept only the IBM format), the source statements are actually reformatted within *FTN. There are two immediate consequences of this arrangement. First, the source listing and any source statement diagnostics produced on SERCOM correspond to the reformatted source module. This should not be a major problem, however, since LINE, EDITED, and LONG formats retain the MTS line numbers so that they may be incorporated into these listings. FORTRAN-H retains line numbers for all formats. Second, the edited source may be obtained by using the EDIT option during the compilation process.

IBM Format

With IBM format, each line is assumed to contain 80 characters. To obtain this number, shorter lines are padded on the right with blanks, while longer lines are truncated on the right after an appropriate error comment has been given on SERCOM.

With IBM format, the optional statement number, consisting of 1 to 5 decimal digits, should be placed within columns 1 through 5 of the first line of the statement; if there is no statement number, these columns must be blank. The source statements are written one per line between columns 1 and 72; however, if a statement is too long for one line it may be continued to a maximum of 19 successive lines by placing a nonzero, nonblank character in column 6 of each such continuation line. Column 6 of the first line of a statement must either be blank or contain the digit zero (0). Columns 73 through 80 of each line are not inspected by the compiler, and though usually employed for program identification and sequencing, may be used for any purpose.

LONG Format

LONG format is identical to IBM format except that there is no sequence ID field and the source statement may continue beyond column 72. The maximum length of one line is 1300 characters. Lines may be continued in the usual manner by using column 6 as the continuation indicator; however, the LINE format and MTS continuation conventions are not recognized. Lines exceeding 72 characters will be reformatted to the IBM format using continuation lines for statements and additional COMMENT lines for comments (the continuation character used will be "#"). Strings should not be split across lines as this may lead to erroneous results. *FTN treats all lines as if they were padded on the right with enough blanks to create an integral number of IBM-format lines after reformatting.

LINE Format

With LINE format, there are no restrictions on the length of the source lines and the entire line is the object of the editing process, i.e., no field comparable to the usual identification field is available.

Comment lines are denoted by a quotation mark (") in the first position. If the last nonblank character of a comment line is a percent sign (%), the line is presumed to be continued on the next line, beginning with the first character of that line. The percent sign and all trailing blanks are ignored. When comment lines are reformatted, a reasonable attempt is made to break them at a blank.

A statement is presumed to be labeled if the first nonblank character of the first line of the statement is numeric. The statement number is interpreted to consist of this first numeric character together with all subsequent numeric characters up to the first nonnumeric character. If the resulting label consists of more than 5 decimal digits, the line, together with its MTS line number, is printed on SERCOM followed by the diagnostic message

LABEL EXCEEDS 5 DIGITS TRUNCATED ON THE RIGHT

and the sixth and all subsequent digits are ignored.

If a statement is labeled, it begins with the first nonblank character following the statement number; otherwise, it begins with the first nonblank character of the line. If the last nonblank character of a statement line is a percent sign (%), the statement is presumed to continue on the next line, beginning with the first character of that line. The percent sign and any trailing blanks are ignored. The total length of a LINE format statement should not exceed 1300 characters.

EDITED Format

With EDITED format, a series of tests is used to determine if the current line is to be edited according to LINE format or passed directly to the compiler as in the IBM format. The following conditions are tested in the sequence given:

(1) If the previous line was processed in LINE format and the last nonblank character of the line was a percent sign, then the current line is also processed in LINE format.

(2) If the line contains more than 72 characters and there are nonblank characters beyond position 72, the line is processed in LINE format. Either a C or a quotation mark in the first position treats the line as a comment line.

(3) If the first character of the line is a C, the line is passed directly to the compiler.

(4) If positions 1 through 5 are blank, the line is passed directly to the compiler. The sixth character of the line is not examined.

(5) If a nonblank, nonnumeric character is found in positions 1 through 5, the line is processed in LINE format.

(6) If positions 1 through 5 contain only blanks or numeric characters and the sixth character is neither a blank nor a zero, the line is processed in LINE format.

(7) If none of the above conditions are true, the line is passed directly to the compiler.

Note that a source module in IBM format will be correctly compiled in EDITED format only if the identification field, positions 73 through 80, of each line is blank. A source module in LINE format may not necessarily compile correctly in the EDITED format. For example, the source lines

```
99999 PAUSE 'THIS WILL NOT %

COMPILE IN EDITED FORMAT'
```

will not be correctly interpreted in EDITED format since the first line appears to correspond to IBM format. Consequently, the LINE format continuation convention is not recognized. In this particular example, the second line is taken as a comment in IBM format. The problem with the EDITED format occurs whenever the first six characters of a source line correspond to IBM format.

BATCH EXAMPLES

In the batch examples given below, commands and compiler options are deliberately not abbreviated so that the user may more readily understand them. The examples which actually contain illustrative programs use the EDITED statement format available by default. The left margin is used to represent column 1 of the input cards.

Example 1

This example illustrates the very common compile-and-execute situation. The program performs the simple function of reading two numbers, forming their product, and printing the results.

```
$RUN *FTN
10   READ 100,A,B
100 FORMAT (2F10.4)
PROD=A*B
PRINT 200,A,B,PROD
200 FORMAT ('0',F8.4,' TIMES ',F8.4,' = ',F15.8)
GO TO 10
END
$RUN -LOAD
2,2/
.111,.222/
```

The following lines are a portion of the output produced by the preceding input program.

```
$RUN -LOAD
EXECUTION BEGINS
  2.0000 TIMES   2.0000 =      4.00000000
  0.1110 TIMES   0.2220 =      0.02464198
```

Example 2

This example is essentially equivalent to the preceding example except that the program has been recoded to form a main program, two subroutines, and a function subprogram. An OPTIONS statement is used in each subprogram to selectively obtain storage maps and object listings. Note that the OPTIONS statement is only valid with the G compiler.

```
$RUN *FTN
OPTIONS: NAME=PROGRAM
10 CALL READER(A,B)
F=FUNCT(A,B)
 CALL WRITER(A,B,F)
```

If this statement had begun in the first column, it would have been interpreted as a comment.

```
GO TO 10
END
OPTIONS:  NAME=READER,SML
SUBROUTINE READER(U,V)
READ 100,U,V
100 FORMAT (2F10.4)
RETURN
END
OPTIONS: NAME=FUNCT,NOSML,SM
```

The occurrence of NOSML disables all listing options, while the subsequent SM reenables the SOURCE and MAP options. This ensures that no unwanted listing options are enabled. The specifications of NOL and SM are equivalent. Note that since the previous OPTIONS statement in the READER subroutine enabled the LIST option, it would remain enabled unless explicitly disabled in this OPTIONS statement.

```
FUNCTION FUNCT(A,B)
FUNCT=A*B
RETURN
END
OPTIONS: NAME=WRITER,NOSML,S
SUBROUTINE WRITER (X,Y,Z)
PRINT 100,X,Y,Z
100 FORMAT (' FUNCTION(',2F8.4,') = ',F15.8)
RETURN
END
$RUN -LOAD
2,2.
.111,.222
```

The following is a portion of the source listing produced for the subroutine WRITER.

```
MICHIGAN TERMINAL SYSTEM FORTRAN IV COMPILER        WRITER

               C     OPTIONS:  NAME=WRITER,NOSML,S  20.000
  0001               SUBROUTINE WRITER (X,Y,Z)      21.000
  0002               PRINT 100,X,Y,Z                22.000
```

The OPTIONS statement is not deleted, instead it is incorporated into the source listing as a comment. The numbers on the left side of the listing are the internal statement numbers (ISN). The numbers on the right side are the MTS line numbers, which are retained when the source statement format is either LINE or EDITED. If two line numbers appear on the right side, then the first character of the statement came from the first line number, and the last character of the statement from the second. The number of intermediate lines and the location of the breaks is not retained.

Example 3

If, in the previous example, the $RUN *FTN command is replaced by the command

```
$RUN *FTN PAR=NOSML
```

and nothing else is changed, then the NOSML and S options in the OPTIONS statements are ignored. The explanation for this follows: In Example 2, the PRINT assignment is defaulted to *SINK* because SOURCE is a default option, and hence it is expected that the PRINT data set will be required. Accordingly, any output listing may be requested in an OPTIONS statement because a PRINT data set is available to handle the output. In Example 3, the interface presumes that the PRINT data set is not needed because of the NOSML option. Consequently, even though the subsequent OPTIONS statements request various output listings, in the absence of a PRINT data set the interface ignores any output listing lines produced by the compiler.

This particular aspect of *FTN may at first seem somewhat peculiar, but it does have some redeeming value. Specifically, it allows the $RUN command options to override any embedded OPTIONS statements. For example, since a DECK data set is not allocated by default, the appearance of the DECK option on embedded OPTIONS statements is ignored. This situation evolved because the OPTIONS statement was added very late in the development of *FORTRANG and *FTN, and *FTN was constructed in such a manner that dynamic allocation of data sets would be extremely awkward. Further, it was felt that the ability to override OPTIONS statements as described above might in fact prove to be more advantageous than dynamic allocation of the data sets.

Example 4

This example is also related to Example 2. It is presumed that a file, MYFILE, exists and contains old, unneeded information. Also, for one reason or another, the user desires to save PROGRM, READER, FUNCT, and WRITER in this file. This can be accomplished as follows.

```
$EMPTY MYFILE
$COPY *SOURCE* MYFILE
```

Between this command and the one that follows, the four source decks for PROGRM, READER, FUNCT, and WRITER should be inserted.

```
$ENDFILE
$RUN *FTN SCARDS=MYFILE
$RUN -LOAD
```

This command sequence would give the desired result. The file MYFILE will contain an exact copy of the source modules.

Example 5

The following RUN command would be used to invoke the FORTRAN-H compiler at optimization level 2.

```
$RUN *FTN SCARDS=MYFILE SPUNCH=OBJECT PAR=OPT=H
```

The object module is placed in the file OBJECT as a result of this command.

CONVERSATIONAL EXAMPLES

In these conversational examples, commands and interface options are generally abbreviated. The left margin represents the left margin of the terminal, and hence generally contains the prefix character which is printed by the system. For example, the pound sign (#) generally is a request for the next command. All characters printed by the system are in uppercase, while all user-entered characters are in lowercase. The notation "(eol)" represents the end-of-line sequence, e.g., carriage return for most terminals. The notation "(attn)" represents an attention-interrupt signal.

Example 1

Although *FTN can read the source statements from the terminal, the procedure is more expensive than placing the source module in a file and then compiling from the file. The following example illustrates a short program used to test new versions of the elementary function routines in the FORTRAN library.

```
#create tester
#FILE "TESTER" HAS BEEN CREATED.
#edit tester
:insert 1
?rewind 9
?namelist /in/ n,a,b,check
?logical check
?10 read (5,in)
?h=(b-a)/(n-1)
?x=a
?if (chcek) goto 30
?c generate the table
?do 20 i=1,n
?y=sqrt(x)
?write (9) x,y
?20 x=x+h
?go to 10
?c generate and check the results
?30 do 32 i=1,n
?y=sqrt(x)
```

```
?read (9) cx,cy
?if (cx.ne.x) pause 'wrong args'
?if (cy.ne.y) print 100,x,y,cy
?32 x=x+h
?go to 10
?c
?100 format (f15.6,2(4x,z8))
?end
?$endfile
:mts
```

In the preceding lines, the MTS file editor was used to enter the source program into the file TESTER.

```
#r *ftn scards=tester
#EXECUTION BEGINS

8.000           IF (CHCEK) GO TO 30
                    $
 *01) SYNTAX*

1 SERIOUS ERROR IN MAIN, NO OBJECT GENERATED.
```

The syntax error occurs because the variable CHCEK is entered as REAL. Thus, the logical IF is erroneously entered as an arithmetic IF and hence the syntax error.

```
#EXECUTION TERMINATED
#edit tester
:a 8 'chcek'check'
:      8      IF (CHECK) GO TO 30
:mts
#r *ftn scards=tester
#EXECUTION BEGINS
 NO ERRORS IN MAIN
#EXECUTION TERMINATED
#r -load 9=-test
#EXECUTION BEGINS
  &in a=1,b=1000,n=1000,check=f &end
$endfile
#EXECUTION TERMINATED
#r -load+-p 9=-test
```

The file -P contains a new version of the SQRT routine. On the previous run, the standard SQRT routine would be loaded from *LIBRARY.

```
#EXECUTION BEGINS
  &in a=1,b=1000,n=1000,check=t &end
       3.000000    411BB67B    411BB67A
       8.000000    412D413D    412D413C
#EXECUTION TERMINATED
```

Example 2

This example illustrates how the user might use the EDIT facility to obtain a copy of a short program being entered directly from the terminal. This technique has the advantage that the whole program need not be reentered just because of a typing error.

```
#r *ftn 1=-s par=edit
```

The EDIT option is assigned so that output will be written to a line file for ease in future correction.

```
#EXECUTION BEGINS
?real*8 x,y
?10 read (5,100)x
?100 format (f20.0)
?y=dsqrt(x)
?priny 200,y
?print 200,y
```

The user sees the typing error, and enters the correct statement. In this case, no additional errors will occur due to the duplication.

```
?200 format (4x,z16)
          5.000            PRINY 200,Y
                            $
 *01) SYNTAX*
?go to 10
?end
?(eol)
   1 SERIOUS ERROR IN MAIN, NO OBJECT GENERATED
#EXECUTION TERMINATED
#edit -s
:delete 5
```

This deletes line number 5 in the file -S.

```
:stop
#r *ftn scards=-s
#EXECUTION BEGINS
  NO ERRORS IN MAIN
#EXECUTION TERMINATED
#r -load
#EXECUTION BEGINS
 3/
    411BB67AE8584CAA
```

Example 3

This example illustrates some of the interface error comments associated with option scanning.

```
#r *ftn scards=aaaa sprint=list par=glitch
#EXECUTION BEGINS
 "GLITCH"?
  AAAA DOES NOT EXIST
  ENTER REPLACEMENT OR CANCEL
 >ccid:notro
  CCID:NOTRO CANNOT BE READ
  ENTER REPLACEMENT OR CANCEL
 >ccid:notro.s
  LIST DOES NOT EXIST
  ENTER REPLACEMENT OR CANCEL
 >cancel
#create list
```

When "cancel" was entered, *FTN returned to MTS command mode. The user then created the file missing file LIST. By $RESTARTING *FTN, the user is again prompted for the replacement.

```
#restart
  ENTER REPLACEMENT OR CANCEL
 >list
 :par=nosml
```

*FTN prompts for options because the user entered the erroneous option "GLITCH" on the $RUN command.

```
?$run -load
 NO ERRORS IN MAIN
#$RUN -LOAD
```

This echo of the command is given by the system when it is requested to execute the command.

```
#EXECUTION BEGINS
 THERE
```

Example 4

The two LINE format editor error comments which may be produced are illustrated in this example. These comments may be produced when the source statement format is either LINE or EDITED.

```
#r *ftn
#EXECUTION BEGINS
?123456789 a=b
      2.000 123456789 A=B
    EDITOR:LABEL EXCEEDS 5 DIGITS.  TRUNCATED ON THE RIGHT
?123
      3.000 123
    EDITOR:  LABELED NULL STATEMENT?  LINE IGNORED
?(attn)
#ATTENTION INTERRUPT AT xxxxxxxx
```

In response to the attention interrupt, MTS receives control in the usual way.

APPENDIX A: INPUT/OUTPUT USING ASSIGNMENT OPTIONS

This appendix is included for the benefit of users who are using old card decks or source files with *FTN. The features described herein are no longer recommended for general use and may be discontinued at some future time. They are included here for reference only.

The following assignment options may be used for input/output to *FTN.

DECK=FDname

If the generated object modules are desired in object deck format, the appropriate FDname may be assigned to this option. Generally, the logical I/O unit 0 is used for this purpose. Use of the DECK assignment option enables the corresponding simple option DECK. The default FDname is the file -DECK which is emptied before use.

EDIT=FDname

If an edited source module is desired, this option may be assigned to an FDname. Use of the EDIT assignment option enables the corresponding simple option EDIT. If the EDIT simple option is specified but the EDIT assignment option is not assigned to a file/device and logical I/O unit 1 was not assigned on the $RUN command, *PUNCH* is used as the default for the batch user and the file -EDIT is used as the default for the conversational user. -EDIT is emptied prior to use.

LOAD=FDname

The LOAD assignment option may be assigned the FDname corresponding to the object module output. Use of the LOAD assignment option enables the corresponding simple option LOAD. If the LOAD option is not assigned to a file or device, the logical I/O unit SPUNCH is used for the LOAD output.

PRINT=FDname

The PRINT assignment option may be assigned the FDname corresponding to the various compiler output listings available through the options SOURCE, MAP, and LIST, or their abbreviated combinations, SM and SML. Use of the PRINT assignment option enables the simple option SOURCE. It is assumed that these output listings will eventually be printed on a device providing at least a 120-character line. No method for adjusting the output line length is provided. The format and content of these listings are given in the FORTRAN-G and FORTRAN-H descriptions in this volume. If PRINT is not assigned to a file or device, the logical I/O unit SPRINT is used for the PRINT data set.

SOURCE=FDname

The SOURCE assignment option may be assigned the FDname corresponding to the compiler input stream. If SOURCE is not assigned to a file or device, the logical I/O unit SCARDS is used to obtain the compiler input.

Input/Output Modifiers

Five special FDname modifiers are provided by *FTN to assist the user in initializing and managing files and devices. These modifiers are EMP, REW, LRECL, BLKSIZE, and RECFM. Since these modifiers are special only to *FTN and are not valid MTS I/O FDname modifiers, they may be used only with FDnames which are specified by assignment options. They are invalid on FDnames which are assigned by MTS logical I/O units.

The special FDname modifiers EMP (empty) and REW (rewind) may be used to initially empty or rewind a file or device. These initialization modifiers may be applied to any FDname; however, EMP is ignored unless it is attached to an output file, and REW is ignored unless it is applied to a rewindable device. These modifiers may be negated by prefixing "NO", not "¬", or minus "-" to the modifier name. If both the positive and negative of an initialization modifier are specified, then the default initialization is performed. Default initialization is NOEMP and NOREW for all files except temporary output files which are being used by default. Such temporary files are emptied by default.

The three special modifiers, RECFM, LRECL, and BLKSIZE, serve the same functions that the corresponding mnemonics do in the IBM Operating System Job Control Language (JCL). RECFM stands for record format, LRECL for logical record length, and BLKSIZE for block size. The permissible values for these options are dependent upon the input/output keyword name and the data set characteristics, and are more fully discussed below. The values assigned LRECL and BLKSIZE must be decimal integers, while the permissible values for RECFM are F (fixed), FB (fixed, blocked), V (variable), VB (variable, blocked), and U (undefined). If an FDname modifier is not recognized, or one of these keywords is assigned a value other than those prescribed above, the FDname will be listed as not existing. For example (user input is in lowercase),

```
#$run *ftn par=source=*tape*@rew@recfm=fba
#EXECUTION BEGINS
 ILLEGAL FDNAME MODIFIER
 *TAPE*@RECFM=FBA DOES NOT EXIST
 RE-ASSIGN SOURCE
```

Note that the legal extended modifier has been removed from the file or device name, leaving only the illegal extended modifier. Also, if the MTS magnetic tape routines are being used to block or deblock the tape, these modifiers should not be used.

Use of the special modifiers described above (REW, EMP, LRECL, BLKSIZE, RECFM) in conjunction with either implicit or explicit concatenation may cause problems.

October 1983

FORTRAN G

Each of the two public files, *FORTRANG and *FTNGTEST, contains a version of the IBM Operating System/360/370 FORTRAN-G compiler. These versions of the compiler were designed for use as subroutines by programs to compile FORTRAN source modules which they generate. Although these compilers may be used as stand-alone processors invoked via a $RUN command, it is recommended that the general user utilize the *FTN interface instead. Since the compilers were not intended to be used as stand-alone processors, they do not provide many of the rudimentary user services expected, e.g., SPUNCH is not defaulted and the compiler options are not dependent on whether the job is batch or conversational. A description of *FTN is given in the section "*FTN Interface" in this volume.

This section is a description of the compilers and their use. It is not a description of the FORTRAN-IV G language. For such a description, see the IBM publication, System/360 and System/370 FORTRAN IV Language, form GC28-6515. The compilers conform mostly to the language as described in that publication, but there are restrictions and extensions of the standard language which are noted in later parts of this section concerning I/O, programming considerations, and miscellaneous FORTRAN features.

These versions of the compiler assume that the input stream contains only FORTRAN source statements in the standard IBM format. Each card image should consist of 80 characters. The first 72 characters are scanned for statements; columns 73-80 are ignored. The compiler options may be dynamically altered during compilation via the use of a pseudo-FORTRAN statement. The OPTIONS statement provides the ability to selectively request various options without having these options in effect for all the source modules compiled.

The difference between the two versions of the G compiler lies in the object modules that they produce. The version of the compiler in *FORTRANG produces a standard object module containing ESD, TXT, RLD, and END records. The *FTNGTEST version produces an object module which includes also SYM records. These SYM records allow the resulting object module to be debugged with the Symbolic Debugging System (SDS). The *FTNGTEST version is slower than the standard version and requires more virtual memory. However, use of this version and SDS can greatly simplify the debugging of programs. A short description of the SDS for FORTRAN users is given in the section "Introduction to Debug Mode for FORTRAN."

The SDS debug facility is not the same as that internal to the FORTRAN-IV compilers. A description of the internal debug facility, which allows the programmer to check for arrays exceeding bounds and to

trace flow of control through the program, is given in the subsection "The FORTRAN Debug Facility."

The following subsections apply to both versions of the G compiler unless otherwise noted. In addition, all references to the IBM System/360 also refer to the IBM System/370, unless otherwise noted.

COMPILER OPTIONS

The compiler options allow the user to control various compiler functions such as which output listings should be produced, which object module format should be used, and what action should be taken if serious compilation errors are discovered. The options are passed to the compiler when it is called, but may be subsequently altered by the occurrence of one or more OPTIONS statements in the input stream. Unrecognizable options are ignored.

Simple options may be negated by prefixing them with either "NO", "¬", or "-". The options may be specified in any order and must be separated by blanks or commas. Option names may be abbreviated by truncation from the right. The following table gives the minimum acceptable abbreviations and the defaults used if the option is not specified. Note that the functions of these options and their defaults are not necessarily the same if the *FTN interface is used. When using *FTN, see the section "*FTN Interface" for the appropriate defaults.

Simple Option	Shortest Abbreviation	Default Value
BCD	B	NOBCD (EBCDIC)
COND	C	COND
DECK	D	DECK
ID	I	ID
LIB	LIB	NOLIB
LIST	L	NOLIST
LOAD	LO	NOLOAD
MAP	M	NOMAP
QUIT	Q	NOQUIT
SCAN	SC	NOSCAN
SM	SM	NOSM
SML	SML	NOSML
SOURCE	S	SOURCE

Assignment Option	Shortest Abbreviation	Default Value
LINE	L	57
NAME	N	MAIN
SIZE	S	4

Simple Option Descriptions

BCD or EBCDIC

The BCD option indicates that the source module lines have been coded in binary coded decimal (as on the 026 keypunch). The standard for the System/370 is Extended Binary Coded Decimal Interchange Code (EBCDIC). Most terminals and the 029 keypunch produce EBCDIC code. If the BCD option is specified, statement numbers passed as arguments must be coded as $n and $ must not be used as an alphabetic character. With EBCDIC, statement numbers in argument lists would be coded &n, so that $ would be a legitimate alphabetic character. The default is EBCDIC.

The compilers do not support BCD characters in either literal data or as print control characters; such characters are treated as EBCDIC. Consequently, for example, a BCD +, used as a carriage-control character will not cause printing to continue on the same line. Programs keypunched in BCD should be carefully scanned for possible errors relating to print control characters and literal data.

COND

The COND option specifies that compilation is to be terminated without producing an object module if serious errors (those with severity levels of 4 or 8) are found in the source program. The default is COND.

DECK

The DECK option specifies that object modules are to be produced by the compiler and that an identification field is to be generated for positions 73-80 of each object module record. The identification field consists of the first four characters of the module name, while the last four are sequentially numbered 0001, 0002, ... The default is DECK.

ID

The ID option generates internal statement numbers (ISN) following external function references in the object code produced by the compiler. (An internal statement number is that number which would be attached to the statement if each executable statement were numbered sequentially from the beginning of the program.) In the generated code, each BALR (branch) to an external program is followed by a four-byte no-operation with an address field equal to the ISN of the source statement containing the external reference. For example, for a call from statement 1000, the four bytes would appear in a hexadecimal dump as 470003E8 (3E8 of base 16 = 1000 of

base 10). This is useful as a debugging aid, but it is not necessary when debugging with SDS. The default is ID.

LIB

The LIB option specifies that each object module generated by the compiler is to be preceded by a LIB record containing its module name. This option affects both the LOAD and DECK output. The LIB record immediately precedes the first ESD record of the object module. The format of the LIB record is as follows:

Columns	Contents
2-4	LIB
17-24	the module name (1 to 8 characters).

For main programs the module name is determined by the value of the NAME assignment option. For subroutines it is always the name given in the FUNCTION or SUBROUTINE statement. The default is NOLIB.

LIST

The LIST option includes a pseudo-assembly-language format listing of the generated object module in the output listings. This listing consists of six columns labeled LOCATION, STA NUM, LABEL, OP, OPERAND, and BCD OPERAND. LOCATION refers to the hexadecimal address of the machine instruction relative to the beginning of the program. STA NUM refers to the first instruction generated for the FORTRAN statement with the indicated ISN. LABEL refers to the FORTRAN statement numbers and compiler-generated statement labels. The OP and OPERAND columns represent the actual machine instruction generated, while the BCD OPERAND attempts to give symbolic interpretation to any variable referenced by the instruction. Unless one understands some machine code and can read hexadecimal dumps, this listing is generally not useful and is in all cases expensive to obtain. The use of the SCAN option overrides the LIST option as there is no object module produced when that option is employed. The COND option can prevent production of the object module and hence the listing. The default is NOLIST.

LOAD

The LOAD option specifies that object modules are to be produced by the compiler and included in the object module output. LOAD has no effect if used with the SCAN option. If the DECK option is specified, each record will be 80 characters long; otherwise, the object records will be 72 characters long. The default is NOLOAD.

Object module lines contain a 12-2-9 punch in the first column and the characters ESD, TXT, RLD, or END, in columns 2-4. The compiler generates four types of ESD items: type 0 items contain the module name, entry point, and module length; type 1 items contain the

entry point names corresponding to ENTRY statements; type 2 items contain the external references made in CALL or EXTERNAL statements and implicit or explicit function references; and type 5 items contain the names for each COMMON block. The FORTRAN-G compilers do not generate type 3 or type 4 ESD items. The TXT records contain user- and compiler-generated constants, translated FORMAT statements, and the generated machine instructions. The information contained in the RLD records is used by the loader to complete external references. External references are resolved by adjusting the constant pointed to by the address in the RLD item by the address of the appropriate external symbol contained in one of the type 2 ESD items. The END record for each module is described below:

Columns	Contents
1	12-2-9
2-4	END
37-39	FTN
41-48	Module name
49-56	Date as MM-DD-YY
57-64	Time as HH:MM.SS
65-68	Number of warning errors
69-72	Number of serious errors

If the DECK option is specified, then each object module record is suffixed with the first four characters of the module name in columns 73-76 and the line sequence number in columns 77-80.

Any LIB records generated will contain the characters LIB in columns 2-4 and the module name in columns 17-24.

Note that when the SCAN option is specified, no punched output will be generated regardless of the LOAD, DECK, or LIB option specifications. Similarly, if the COND option is specified, then no punched output will be generated for any source module containing serious errors, i.e., diagnostics with severity level 4 or 8.

MAP

The MAP option specifies that a storage map is to be included in the output listings. The map produced consists of tables containing variable names and locations for common, EQUIVALENCE, NAMELIST, scalar and array variables, subprograms referenced, and FORMAT statements. The default is NOMAP.

QUIT

The QUIT option terminates the job if there are serious compilation errors detected in one or more of the source modules or if a fatal compiler error occurs. The compiler terminates the job by calling the QUIT subroutine (see MTS Volume 3, System Subroutine Descriptions, for a description of this subroutine). The QUIT option is

effective for both batch and conversational users. The QUIT subroutine does not immediately terminate the job but sets a flag so that when the job returns to MTS command mode, it will be terminated, i.e., signed off. If the compiler is being used as a stand-alone language processor, then calling QUIT is tantamount to terminating the job, since immediately after calling QUIT the compiler returns to its caller, MTS. When the compiler is being used as a subroutine, job termination is dependent on the actions taken by the calling program. The default is NOQUIT.

SCAN

The SCAN option causes the compiler to scan the source module for syntax and compilation errors. All appropriate error diagnostics are generated for each source module, but no object code is generated and consequently neither the object listing nor the program size can be included in the output listings. In addition, no object module output is produced. Use of the SCAN option will decrease the cost of scanning the source program for errors. The default is NOSCAN.

SM

The SM option is the abbreviation for SOURCE and MAP combined.

SML

The SML option is the abbreviation for SOURCE, MAP, and LIST combined.

SOURCE

The SOURCE option produces a listing of the source deck in the output listings. If NOSOURCE is specified and source errors are found, the source statement in error and the diagnostic message are still included in the output listings. The default is SOURCE.

Assignment Option Descriptions

LINE=n

The LINE assignment option specifies the number of lines per page for the output listings. "n" must be an integer in the range (3,32767). This may be used to control the spacing of the page headers, etc., as they are only produced at the beginning of each set of lines. The default is 57 lines per page.

NAME=xxxx

The NAME assignment option specifies the name to be used as the module name for all main programs compiled. Subprogram module names are always the name given in the FUNCTION or SUBROUTINE statement. The module name appears in the page headers, LIB records, the ESD type 0 lines, the END lines, and the object code at relative address 000005. The name "xxxx" may be from 1 to 8 characters in length. The default is MAIN.

SIZE=n

The SIZE assignment option specifies the number of pages of virtual memory to be used for compiler working storage. "n" must be an integer in the range of 1 to 255. Internally, the FORTRAN-G compiler allocates all dynamically acquired working storage in page-size (4096 bytes) blocks. In order to avoid problems caused by the assumption that the storage is sequentially available, and to increase efficiency, this has been altered so that the compiler obtains "n" pages from the system and suballocates the remainder on a page basis. If additional storage is required and available, it is again obtained in units of "n" pages. The default is 4 pages.

The OPTIONS Statement

Any source line beginning with the character string

"OPTIONS:"

starting in column 7 is recognized as an OPTIONS statement. The source line must begin with precisely this string; i.e., it cannot be labeled, the letters of the word OPTIONS must be consecutive with no intervening blanks, and the colon must appear in column 14. This rigid set of rules will prevent any valid FORTRAN source line from being mistakenly interpreted as an OPTIONS statement.

When an OPTIONS statement is encountered, columns 15 through 72 inclusive are scanned for valid compiler options. The simple options SCAN, COND, LIB, and QUIT, and the assignment option SIZE are not legal in an OPTIONS statement and are ignored. Following the option scan, the character "C" is placed in column 1 so that the OPTIONS statement will appear in the source listing as a comment line.

The options appearing in an OPTIONS statement modify the existing options; they do not replace them. These modified options will take effect with the first source module which follows the OPTIONS statement, and remain in effect until another OPTIONS statement is encountered. If an OPTIONS statement is the first source line of a module, i.e., it precedes all source statements and comments, then the modified options will be effective for that module. If an OPTIONS statement follows the

first comment or source line of a module, then the modified options will not take effect until the next source module is encountered. Since the modified options are retained until the next OPTIONS statement, care should be used in ordering the source modules unless each source module is preceded by an OPTIONS statement which completely specifies the options desired for that source module. This is particularly appropriate if the LIST option is being changed.

USE OF FORTRAN-G AS A SUBROUTINE

Both the standard version and the test version of the compiler may be called as a subroutine. The compiler to be called must be concatenated to the object (calling) program on the $RUN command, e.g.,

```
$RUN object+*FORTRANG [logical I/O units]
```

or

```
$RUN object+*FTNGTEST [logical I/O units]
```

Each compiler is called using the same calling sequence and the same arguments. The calling sequence is

```
CALL FTNG(options,reader,printer,punch,break,&rc4,...,&rc32)
```

All of the arguments except the first are optional. The calling sequence must conform to the FORTRAN standard described in MTS Volume 3, System Subroutine Descriptions. Specifically, since most of the arguments are not required, the high-order byte of the last argument address must contain X'80'. Although only the first argument is required, in order to specify one of the optional arguments, all of the preceding arguments must also be specified, i.e., if one wanted to specify the printer, he would also have to specify the reader, e.g.,

```
CALL FTNG(options,reader,printer,&rc4,...,&rc32)
```

All of the optional arguments are subroutine names and must be declared in an EXTERNAL statement in a FORTRAN program. For assembly language users, these optional arguments are passed by placing a pointer to the appropriate V-type address constant in the parameter list.

The arguments to FTNG are described below:

Options

This is the only mandatory argument and may be specified in either of two ways: as a standard parameter field or in the direct form. The standard parameter field is a halfword character count immediately followed by the parameter characters, e.g.,

```
INTEGER*2 PAR(5)/7,'SM',' Q','UI','T '/
```

The parameters may appear in any order in the field. When given in this manner, the options modify the default options.

If the parameter field length is negative, then the options parameter is the address of an eight (8) halfword vector formatted as follows: four halfwords containing the eight-character default main program name, left-justified with trailing blanks, two halfwords containing the option bits, one halfword containing the number of lines per page, and one halfword containing the memory block size. Within the first option bit halfword, the parameter is enabled if the corresponding bit is set; otherwise, it is disabled. The bit assignments are as follows:

Bit	Hex	Function	Default
0	80000000	MAP	0
1	40000000	LOAD	0
2	20000000	DECK	1
3	10000000	LIST	0
4	08000000	SOURCE	1
5	04000000	BCD	0
6	02000000	ID	1
7	01000000	----	0
8	00800000	----	0
9	00400000	----	0
10	00200000	----	0
11	00100000	----	0
12	00080000	LIB	0
13	00040000	QUIT	0
14	00020000	SCAN	0
15	00010000	COND	1

The remaining bits should be 0 (zero).

The following example from a BLOCK DATA program illustrates how the default options may be set up to be passed in the direct form:

```
COMMON /OPTION/ NAME,OPT,LINE,SIZE
REAL*8 NAME/'MAIN'/
INTEGER*2 OPT(2)/Z2A01,0/,LINE/57/,SIZE/4/
```

Note that when the options are passed directly that they completely replace the default options, rather than modify them. If the options are passed directly, i.e., the high-order bit of the first argument is one, then the options must be completely specified. It is not possible to simply change some of the options bits and pass them directly. Note that the high-order bit of any alphanumeric character is a one; it

is for this reason that the default main program name appears first in the options vector described above.

Reader

This argument is the name of an external function to be called to obtain the source input lines. If this optional argument is not given, the default value is SCARDS. The parameter list is the same as that described for the MTS READ subroutine in MTS Volume 3. The unit described for the READ subroutine is set to logical I/O unit 0. Although there are several parameters in the READ subroutine calling sequence, the user need only be concerned with the first. FTNG passes the address of an 80-character, preblanked buffer, and expects that the source will be placed in the buffer. For example:

```
      SUBROUTINE READER(BUFFER,*)
      REAL*8 BUFFER(10)
         .
         .
      RETURN
  100 RETURN 1
      END
```

Any nonzero return code passed by the reader routine to the compiler will be interpreted as an end-of-file. On an end-of-file indication the buffer contents are ignored. Any attempt to place more than 80 characters in the buffer will result in a fatal compiler error. If fewer than 80 characters are placed in the buffer, then the blanks that are placed in the buffer prior to the call may be used to pad the input line. The set of source modules to be compiled must be in standard IBM format and terminated by an end-of-file condition, i.e., a nonzero return code from the routine.

Printer

This argument is the name of an external function to be called to dispose of the output listing lines produced by the compiler. If the optional argument is not given, a default value of SPRINT is used. The printer parameter list corresponds to that described for the WRITE subroutine in MTS Volume 3. The unit described in that parameter list is set to MTS logical I/O unit 1. Although there are several arguments, the programmer need only be concerned with the first, the 120-character output line.

The printer output line consists of 120-character output lines with the logical carriage-control characters 0, 1, or blank. The amount and type of information is controlled by the options SOURCE, MAP, and LIST. Each source module will

generally produce at least two lines, a page heading containing the module name, date, and time, and a line giving the total memory storage requirements of the program in bytes. Any diagnostics are included regardless of the options requested.

See the descriptions of the SOURCE, MAP, and LIST parameters for a description of the output that is produced for each option.

Punch

This argument is the name of an external function which will be called to dispose of any object module lines produced by the compiler. If the argument is not given, a default value of SPUNCH is supplied. It should be noted that unless one is running a batch job, with a nonzero card estimate, SPUNCH is not defaulted. When defaulted, SPUNCH defaults to *PUNCH*. The parameter list for the call to the punch routine is the same as that for the system subroutine WRITE as described in MTS Volume 3. Unit corresponds to logical I/O unit 2. Of the arguments in the call, only the first two are of importance to a user-supplied routine. These arguments are the buffer containing the object module line and a halfword integer (INTEGER*2) output line length, respectively. The object module lines will be 80 character if the DECK parameter is specified; otherwise, they will be 72. If the LIB parameter is specified, then each module will be preceded by a LIB card.

For a description of object module output see the LOAD, DECK, and LIB parameters. Note that when the SCAN parameter is given, no punched output will be generated regardless of the LOAD, DECK, or LIB parameters. Similarly, if the default parameter COND is not changed, then no punched output will be generated for any source modules containing diagnostics with severity levels 4 or 8.

Break

This argument is the name of an external function to be called after each source module is compiled and before the next compilation has begun; however, because of the way in which the compiler operates, the first line of the next source module will already have been read. If this optional argument is not given, the compiler simply proceeds to the compilation of the next source module.

The argument to the BREAK routine is a six-element integer vector containing the following information:

Word	Information
1-2	The six-character module name of the program just compiled, left-justified with trailing blanks.
3	Current compiler options.
4	Number of severity level 0 diagnostics for this source module.
5	Number of severity level 4 diagnostics for this source module.
6	Number of severity level 8 diagnostics for this source module.

If the BREAK routine gives a nonzero return code to the compiler, it will immediately halt and return to its calling program with the appropriate return code.

Return Codes

The compiler returns in the normal FORTRAN manner (see a description of FORTRAN calling sequences in MTS Volume 3, System Subroutine Descriptions). In addition, the return code is also placed in general register 0 (zero), so that FTNG may be declared an INTEGER*4 function, and as such, will assume the values 0, 4,...,32. The return codes and explanations follow.

Return Code	Meaning
0	All compilations have been completed and no errors were found.
4	All compilations have been completed and only diagnostics given have severity level 0.
8	All compilations have been completed and at least one severity level 4 diagnostic was given.
12	All compilations have been completed and at least one severity level 8 diagnostic was given.
16	Compiler malfunction. Try compiling the source module again and if the error persists, see a Computing Center consultant.
20	Compiler malfunction due to an unanticipated program interrupt. Note that a program interrupt in any of the optional user-supplied routines will manifest itself in this manner. The compiler interrupt processor ignores floating-point overflow and underflow during the first phase of the compilation when the source module is being read. If it is not reading the source module, it immediately

	stops. Program interrupt control may be most easily removed from FTNG by calling PGNTTRP the first time the user input routine is called.
24	Insufficient memory is available for tables. Considering the current size of virtual memory, this problem should never arise.
28	This code is given if a source line containing more than 80 characters is passed to the compiler. Since the compiler supports only the standard IBM format FORTRAN, the input buffer is only 80 characters long. Consequently, if a line in excess of 80 characters is placed in the buffer, the resulting overflow destroys information that cannot be recovered.
32	Insufficient memory is available to buffer the object module being generated.

USE OF THE FORTRAN-G COMPILERS AS STAND-ALONE LANGUAGE PROCESSORS

As indicated earlier, use of the two versions of the FORTRAN-G compiler as stand-alone language processors is not recommended. However, use of *FORTRANG in this manner does provide an efficient batch-oriented interface to the compiler. Conversational users will find this version of the compiler inconvenient to use and should refer to the section "*FTN Interface" in this volume. The following deck structure would be sufficient to compile a set of FORTRAN programs punched on cards according to the standard IBM format.

```
$SIGNON ccid 'name'
password
$RUN *FORTRANG SPUNCH=-LOAD PAR=QUIT
          .
     (source program)
          .
$ENDFILE
$RUN -LOAD
$SIGNOFF
```

The characters "ccid" are the user's four-character Computing Center signon ID. The second card contains the user's password starting in column 1. The QUIT option is specified here so that if the compiler finds any serious errors, the job will be terminated; otherwise, the system would proceed to the second $RUN command and load and execute the erroneous program, and hence increase the cost of the job.

In the second example below, the NOCOND option is specified and the default of NOQUIT is allowed to stand. Thus, regardless of how many errors are found in the FORTRAN program, the compiler will produce an object module that will be loaded and executed. This is not the same as simply using the default options since, by default, if serious errors

were found, no object module would be produced. Thus, in an error situation, the second $RUN command would cause a loader error comment because the file -LOAD would be empty. This example also illustrates the defaulting of the logical I/O units 5 and 6, and one of the more convenient features of formatted input, namely, the use of a comma (,) in a formatted numeric field.

```
$SIGNON ccid 'name'
password
$RUN *FORTRANG SPUNCH=-LOAD PAR=NOCOND
      READ (5,100) I,J
  100 FORMAT (2I4)
      ISUM=I+J
      WRITE (6,200) I,J,ISUM
  200 FORMAT (I4,' +',I4','=',I4)
      END
$ENDFILE
$RUN -LOAD
2,2,
$SIGNOFF
```

This example will produce a single output line of the form

2 + 2 = 4.

This final example serves to illustrate the use of the OPTIONS statement.

```
$SIGNON ccid 'name'
password
$RUN *FORTRANG SPUNCH=-LOAD PAR=QUIT,SM
      OPTIONS:NAME=HEATFLOW,SML
          ...
      END
      OPTIONS:NAME=SUB1,NOSML,SOURCE
          ...
      END
      OPTIONS:NAME=FCN1,NOSML,SM
          ...
      END
$ENDFILE
$RUN -LOAD
```

In the above example, the source programs are omitted except for the END statements. The first program would be compiled with SOURCE, MAP, and LIST, the second with only SOURCE, and the final one with SOURCE and MAP. Note that the SM option given in the PAR field was needlessly specified. Note also that each OPTIONS statement completely specifies the listing options. The initial NOSML turns off all the listing options, while the subsequent options turn back on the desired listings. The NAME option was used so that the correct program name would appear on all the page headers. Frequently, the first page header for a subprogram will use the default main program name instead of the name on

the SUBROUTINE or FUNCTION card, because comment cards precede this card and thus the first page header must be generated before the correct module name is known. Presumably, the NAME option has been assigned the proper module names in this case, though it will cause no problem if it has not been assigned correctly.

FORTRAN-G SOURCE MODULE ERROR/WARNING MESSAGES

The diagnostic messages produced by the compiler occur in the source listing immediately following the source statement to which they refer. The following example illustrates the format of these messages:

```
XX = A+B+-C/(X**3-A**-75)
          $
n) xxx message, n+1) xxx message
```

where

n	is an integer noting the positional occurrence of the error in the line.
xxx	is a three-digit message number of the form IEYxxxI.
$	is the symbol used for flagging the individual errors in the statement. This symbol is placed underneath the character causing the error.

"message" is a cryptic description of the type of error. The error and warning messages are distinguished by the resulting severity levels. Serious error messages have a severity level of 4 to 8, while warning messages have a severity level of 0.

IEY001I ILLEGAL TYPE

This message is associated with a source module statement when the type of a variable is not correct for its usage. Examples of situations in which this message would be given are: (1) the variable in an assigned GO TO statement is not an integer variable; (2) in an assignment statement, the variable to the left of the equal sign is of logical type and the expression to the right is not. Severity Level: 8.

IEY002I LABEL

This message appears with a statement which should be labeled and is not. Examples of such statements are FORMAT statements and statements following GO TO statements. Severity Level: 0.

IEY003I NAME LENGTH

The name of a variable, COMMON block, NAMELIST, or subprogram exceeds six characters in length. If two variable names appear in an expression without a separating operation symbol, this message is produced. Severity Level: 0.

IEY004I COMMA

A comma is supposed to appear in a statement and it does not. Severity Level: 0.

IEY005I ILLEGAL LABEL

The usage of a label is invalid. For example, if an attempt is made to branch to the label of a FORMAT statement, ILLEGAL LABEL is produced. Severity Level: 8.

IEY006I DUPLICATE LABEL

A label appearing in the label field of a statement is already defined (has appeared in the label field of a previous statement). Severity Level: 8.

IEY007I ID CONFLICT

The name of a variable or subprogram is used improperly, in the sense that a previous statement or a previous portion of the present statement has established a type for the name, and the present usage is in conflict with that type. Examples of such situations are: (1) the name listed in a CALL statement is the name of a variable, not a subprogram; (2) a single name appears more than once in the dummy list of a statement function; (3) a name listed in an EXTERNAL statement has already been defined in another context. Severity Level: 8.

IEY008I ALLOCATION

Storage assignments specified by a source module statement cannot be performed due to an inconsistency between the present usage of a variable name and some prior usage of that name, or due to an improper usage of a name when it first occurred in the source module. Examples of the situations causing the error are: (1) a name listed in a COMMON block has been listed in another COMMON block; (2) a variable listed in an EQUIVALENCE statement is followed by more than seven subscripts. Severity Level: 8.

IEY009I ORDER

The statements of a source module are used in an improper sequence. For example, an IMPLICIT statement appears as anything other than the first or second statement of the source module, or an ENTRY statement appears within a DO loop. Severity Level: 8.

IEY010I SIZE

A number used in the source module does not conform to the legal values for its use. Examples are: (1) the size specification in an explicit specification statement is not one of the acceptable values; (2) a label which is used in a statement exceeds the legal size for a statement label; (3) an integer constant is too large. Severity Level: 8.

IEY011I UNDIMENSIONED

A variable name indicates an array (i.e., subscripts follow the name), and the variable has not been dimensioned. Severity Level: 8.

IEY012I SUBSCRIPT

The number of subscripts used in an array reference is either too large or too small for the array. Severity Level: 8.

IEY013I SYNTAX

The statement or part of a statement to which it refers does not conform to FORTRAN-IV syntax. If a statement cannot be identified, this error message is used. Other cases in which it appears are: (1) a nondigit appears in the label field; (2) fewer than three labels follow the expression in an arithmetic IF statement. Severity Level: 8.

IEY014I CONVERT

In a DATA statement or in an explicit specification statement containing data values, the mode of the constant is different from the mode of the variable with which it is associated. The compiler converts the constant to the correct mode. Therefore, this message is simply a notification to the programmer that the conversion is performed. Severity Level: 0.

IEY015I NO END CARD

The source module does not contain an END statement. Severity Level: 0.

IEY016I ILLEGAL STA.

The statement (sta) to which it is attached is invalid in the context in which it has been used. Examples of situations in which this message appears are: (1) the statement in a logical IF statement (the result of the true condition) is a specification statement, a DO statement, etc.; (2) an ENTRY statement appears in the source module and the source module is not a subprogram. Severity Level: 8.

IEY017I ILLEGAL STA. WRN

This is a warning (WRN) message. A RETURN I statement appears in any source module other than a SUBROUTINE subprogram. Severity Level: 0.

IEY018I NUMBER ARG

A reference to a library subprogram appears with the incorrect number of arguments specified. Severity Level: 4.

IEY027I CONTINUATION CARDS DELETED

More than nineteen continuation lines were read for one statement. All subsequent lines are skipped until the beginning of the next statement is encountered. Severity Level: 8.

IEY032I NULL PROGRAM

This error occurs if an end-of-file comes before any identifiable source statement. Severity Level: 0.

IEY033I COMMENTS DELETED

More than thirty comment lines were read between the initial lines of two consecutive statements. The thirty-first comment line and all subsequent comment lines are skipped until the beginning of the next statement is encountered. (There is no restriction on the number of comment lines preceding the first statement.) Severity Level: 0.

IEY036I ILLEGAL LABEL WRN

The label on this nonexecutable statement has no valid use beyond visual identification, and may produce errors in the object module if the same label is the target of a branch-type statement. Only branches to executable statements are valid. This message is produced, for example, when an END statement is labeled. Severity Level: 0.

IEY037I PREVIOUSLY DIMENSIONED WRN

This message appears when there is an attempt to redimension an array. The dimensions given to the first occurrence of the statement are used. The message only occurs if the dimensions are changed. Severity Level: 4.

IEY038I SIZE WRN

An attempt has been made to initialize a variable with a value which exceeds the size of the scalar, array, or array element. The message will occur in the following circumstances:

(1) Five bytes of initializing data are given for a scalar variable: REAL A/'ABCDE'/
(2) Excessive bytes are given for an element of an array: DATA A(1)/'ABCDEFG'/
(3) Use of data spill to initialize an array: DIMENSION ARRAY (3) ARRAY/'ABCDEFGHIJKL'/

The warning is given and the normal FORTRAN procedures are followed, i.e., the variable is initialized with a truncated value. Severity Level: 4.

IEY039I RETURN

A return statement is needed in a subroutine. Severity Level: 0.

IEY045I SP CONSTANT

The constant flagged is typed as single precision in spite of the fact that seven or more digits were coded for it. Severity Level: 0.

IEY046I DP CONSTANT

The constant flagged is typed double precision (REAL*8) because it contains more than 7 digits; the letter D is not specified. Severity Level: 0.

The source module listing, with error indications and error messages for the errors detected during initial processing of the source

statements, is produced by phase 1 of the compiler. Certain program errors can occur, however, which cannot be detected until storage allocation takes place. These errors are detected and reported by phase 2 and are described in the following paragraphs.

IEY019I FUNCTION ENTRIES UNDEFINED

When the program being compiled is a FUNCTION subprogram, a check is made to determine whether a scalar with the same name as the FUNCTION and each ENTRY is defined. If no such scalars are listed on the SCALAR roll, the error message is written on the source module listing. The message is followed by a list of the undefined names. Severity Level: 0.

IEY020I COMMON BLOCK / / ERRORS

Errors of two types can exist in the definitions of EQUIVALENCE sets which refer to the COMMON area. The first type of error exists because of a contradiction in the allocation specified, e.g., the EQUIVALENCE sets(A,B(6), C(2)) and (B(8),C(1)). The second type of error is due to an attempt to extend the beginning of the COMMON area, as in COMMON A,B,C and EQUIVALENCE (A,F(10)).

An additional error in the assignment of COMMON storage occurs if the source program attempts to allocate a variable to a location which does not fall on the appropriate boundary. Since each COMMON block is assumed to begin on a double-precision boundary, this error can be produced by either (or both) the COMMON statement or an EQUIVALENCE statement which refers to COMMON.

When each block of COMMON storage has been allocated, the error message is printed if any error has been detected (the block name is provided). The message is followed by a list of the variables which could not be allocated due to the errors. Severity Level: 4.

IEY021I UNCLOSED DO LOOPS

If DO loops are initiated in the source module, but their terminal statements do not exist, the second phase of the compiler finds pointers to the labels of the nonexistent terminal statements on the DO LOOPS OPEN roll. If pointers are found on the roll, the error message is printed, followed by a list of the labels which appeared in all DO statements that were not defined in the source module. Severity Level: 8.

IEY022I UNDEFINED LABELS

If any labels are used in the source module but are not defined, they constitute label errors. At the conclusion of the check for this situation, the error message is printed. If there are undefined labels used in the source module, they are listed on the lines following the message. Severity Level: 8.

IEY023I EQUIVALENCE ALLOCATION ERRORS

Allocation errors due to the arrangement of EQUIVALENCE statements which do not refer to COMMON variables may have two causes. The first cause is the conflict between two EQUIVALENCE sets; for example, (A,B(6),C(3)) and (B(8),C(1)).

The second cause is incompatible boundary alignment in the EQUIVALENCE set. The first variable in each EQUIVALENCE set is assigned to its appropriate boundary, and a record is kept of the size of the variable. Then, as each variable in the set is processed, if any variable of a greater size requires alignment, the entire set is moved accordingly. If any variable is encountered of the size which caused the last alignment, or of lower size, and that variable is not on the appropriate boundary, this type of an equivalence error has occurred.

If EQUIVALENCE errors of either of these types occur, the error message is printed. The message is followed by a list of the variables which could not be allocated according to source module specifications. Severity Level: 4.

IEY024I EQUIVALENCE DEFINITION ERRORS

Another category of EQUIVALENCE errors is the specification, in an EQUIVALENCE set, of an array element which is outside the array. These errors are summarized under the above error message on the source module listing. Severity Level: 4.

IEY025I DUMMY DIMENSION ERRORS

If variables specified as dummy array dimensions are not in COMMON or not global dummy variables, they constitute errors. These are summarized under the above error message on the source module listing. Severity Level: 4.

IEY026I BLOCK DATA ERRORS

If variables specified within the BLOCK DATA subprogram have not also been defined in a COMMON, they constitute

errors. The error message is produced on the source module listing followed by a sum of the variables in error. Severity Level: 4.

IEY040I COMMON ERROR IN BLOCK DATA

There was an error in a BLOCK DATA subprogram. The BLOCK DATA routine must have at least one named COMMON section and cannot contain references to blank COMMON. Severity Level: 8.

IEY041I COMMON INITIALIZATION ERRORS

An attempt has been made to initialize a variable in blank COMMON or an attempt has been made to initialize a labeled COMMON area outside of a BLOCK DATA subprogram. The variables in error are listed below the message. Severity Level: 8.

THE FORTRAN DEBUG FACILITY

The FORTRAN debug facility is a strictly batch-oriented facility and is only available with the FORTRAN-G compilers. It is neither flexible nor general and users are advised to use the other debugging facilities in MTS instead. The Symbolic Debugging System (SDS) used with the *FTNGTEST compiler is far superior to the FORTRAN debug facility. For program development, users should consider *IF which accepts the same language as FORTRAN G and gives better diagnostic messages. The following paragraphs are reprinted from the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

The FORTRAN debug facility consists of a DEBUG specification statement, an AT debug packet identification statement, and three executable statements. These statements, alone or in combination with any FORTRAN source language statements, are used to state the desired debugging operations for a single program unit in source language. (A program unit is a single main program or a subprogram.)

The source deck arrangement consists of the source language statements that comprise the program, followed by the debug packets, followed by the END statement.

The statements that make up a program debugging operation must be grouped in one or more debug packets. A debug packet is preceded by the AT debug packet identification statement and consists of one or more executable debug facility statements, and/or FORTRAN source language statements. A debug packet is terminated by either another debug packet identification statement or the END statement of the program unit.

Debug Facility Statements

The specification statement (DEBUG) sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking). The AT statement identifies the beginning of the debug packet and the point in the program at which debugging is to begin. The three executable statements (TRACE ON, TRACE OFF, and DISPLAY) designate actions to be taken at specific points in the program. The following text explains each debug facility statement and contains several programming examples.

DEBUG Statement

There must be one DEBUG statement for each program or subprogram to be debugged, and it must immediately precede the first debug packet.

General Form:

```
DEBUG option[,...]
```

where "option" may be any one of the following:

UNIT (dsr)

where "dsr" is an integer constant that represents a FORTRAN logical I/O unit number. All debugging output is written to this I/O unit. If this option is not specified, any debugging output is written to unit 6 (which defaults to *SINK*). All unit definitions within an executable program must refer to the same unit.

SUBCHK [(n1[,n2,...,nn])]

where "n1,n2,...,nn" are array names. The validity of the subscripts used with the named arrays is checked by comparing the subscript combination (by converting it to its one-dimensional equivalent) to the size of the array. If the subscript exceeds its dimension bounds, a message is written to the debug output I/O unit. Note that this will not catch all illegal subscripts of a multidimensioned array, but only those whose converted subscript is larger than the size of the array. Program execution continues, using the illegal subscript. If the list of array names is omitted, all arrays in the program are checked for valid subscript usage. If the entire option is omitted, no arrays are checked for valid subscripts.

TRACE

This option must be in the DEBUG statement of each program or subprogram for which tracing is desired. If

this option is omitted, there can be no display of program flow by statement number within the program. Even when this option is used, a TRACE ON statement must appear in the first debug packet in which tracing is desired.

INIT (m1,m2,...,mn)

where "m1,m2,...,mn" are names of variables or arrays that are to be written to the debug output I/O unit only when the variable or the array values change. If "m1" is a variable name, the name and value are displayed whenever the variable is assigned a new value in either an assignment, READ, or an assigned GO TO statement. If "m1" is an array name, the changed element is displayed. If the list of names is omitted, a display occurs whenever the value of a variable or an array element is changed. If the entire option is omitted, no display occurs when values change.

SUBTRACE

This option specifies that the name of this subprogram is to be displayed whenever it is entered. The message RETURN is to be displayed whenever execution of this subprogram is completed.

The options in a DEBUG statement may be given in any order and they must be separated by commas.

AT Statement

The AT statement identifies the beginning of a debug packet and indicates the point in the program at which debugging is to begin. There must be one AT statement for each debug packet; there may be many debug packets for one program or subprogram.

General Form:

AT n

where "n" is an executable statement number in the program or subprogram to be debugged.

The debugging operations specified within the debug packet are performed prior to the execution of the statement indicated by the statement number in the AT statement.

TRACE ON Statement

The TRACE ON statement initiates the display of program flow by statement number. Each time a statement with an external

statement number is executed, a record of the statement number is written to the debug output I/O unit. This statement has no effect unless the TRACE option was specified in the DEBUG statement.

General Form:

TRACE ON

For a given debug packet, the TRACE ON statement takes effect immediately before the execution of the statement specified in the AT statement; tracing continues until a TRACE OFF statement is encountered. The TRACE ON stays in effect through any level of subprogram call or return. However, if a TRACE ON statement is in effect and control is given to a subprogram in which the TRACE option was not specified, the statement numbers in that program are not traced. Trace output is written to the debug output I/O unit.

This statement may not appear as the conditional part of a logical IF statement.

TRACE OFF Statement

The TRACE OFF statement may appear anywhere within a debug packet and stops the recording of program flow by statement number.

General Form:

TRACE OFF

This statement may not appear as the conditional part of a logical IF statement.

DISPLAY Statement

The DISPLAY statement may appear anywhere within a debug packet and causes data to be displayed in NAMELIST output format.

General Form:

DISPLAY list

where "list" is a series of variable or array names, separated by commas.

The DISPLAY statement eliminates the need for FORMAT or NAMELIST and WRITE statements to display the results of a debugging operation. The data are written to the debug output I/O unit.

The effect of a DISPLAY statement is the same as the following FORTRAN source language statements:

```
NAMELIST /name/list
WRITE (n,name)
```

where "name" is the same in both statements. Note that array elements may not appear in the list. If the DISPLAY statement appears in a SUBROUTINE subprogram, a dummy argument may not be included in the list. This statement may not appear as the conditional part of a logical IF statement.

Programming Considerations

The following precautions must be taken when setting up a debug packet:

(1) Any DO loops initiated within a debug packet must be wholly contained within that packet.

(2) Statement numbers within a debug packet must be unique. They must be different from statement numbers within other debug packets and within the program being debugged.

(3) An error in a program should not be corrected with a debug packet; when the debug packet is removed, the error remains in the program.

(4) The following statements must not appear in a debug packet:

```
SUBROUTINE
FUNCTION
ENTRY
IMPLICIT
BLOCK DATA
statement function definition
```

(5) The program being debugged must not transfer control to any statement number defined in a debug packet; however, control may be returned to any point in the program from a packet. In addition, a debug packet may contain a RETURN, STOP, or CALL EXIT statement.

The FORTRAN internal debugging facility may provide a useful debugging aid for batch programmers. Those interested in interactive debugging capabilities should note the section on the Symbolic Debugging System (SDS). SDS provides many debugging aids including those of the Debug Facility.

October 1983

FORTRAN H

INTRODUCTION

The public file *FORTRANH contains a version of the IBM Operating System/360 FORTRAN-H compiler modified to run under MTS. The compiler contained in *FORTRANH (like the two versions of the FORTRAN-IV G compiler contained in *FORTRANG and *FTNGTEST) is really meant to be used as a subroutine by programs to compile FORTRAN source modules. Although this compiler may be used as a stand-alone language processor invoked via a $RUN *FORTRANH command, this is not recommended.

Users wishing to utilize the capabilities of the FORTRAN-H compiler should use the *FTN interface (see the section "*FTN Interface" in this volume). *FTN provides many services such as defaulting logical I/O units, conversational entry and correction of options, etc., which are not provided by the simplified compiler in *FORTRANH.

The primary advantage of using FORTRAN-H instead of FORTRAN-G is that FORTRAN-H generates a program which executes more efficiently if the optimization features are used. However, the FORTRAN-H compiler provides only limited set of debugging facilities:

(1) a limited set of SYM records are generated for use with the Symbolic Debugging System (SDS), and

(2) FORTRAN-H does not support the internal debug feature described in the section "FORTRAN G."

Compilation of a program with the FORTRAN-H compiler can cost significantly more than compilation with either of the FORTRAN-G compilers. It is recommended that initial program debugging be done using the FORTRAN-G compiler and that the final production version be compiled using the FORTRAN-H compiler, if the optimization features are desired.

This section is not a description of the FORTRAN-IV language. For such a description, see the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

COMPILER OPTIONS

The compiler options allow the user to control various compiler functions such as what output listings should be produced, which object module format should be used, etc. Unrecognizable options are ignored.

Negation of simple options is accomplished by prefixing them with "NO", "-", or "¬". The options must be separated by blanks or commas. The option names may be abbreviated by truncation from the right. The table below gives the minimum acceptable abbreviations and the default values used if the option is not specified.

Simple Option	Shortest Abbreviation	Default Value
BCD	B	NOBCD (EBCDIC)
COMMENT	C	COMMENT
DECK	D	DECK
EJECT	EJ	EJECT
ERR	ER	ERR
ID	ID	NOID
LIST	L	NOLIST
LOAD	LO	NOLOAD
MAP	M	MAP
PRINT	P	PRINT
SCAN	SC	NOSCAN
SOURCE	S	SOURCE
STRUC	ST	NOSTRUC
TEST	T	NOTEST
XL	XL	NOXL
XREF	X	XREF

Assignment Option	Shortest Abbreviation	Default Value
CALIGN	CA	0
CSHIFT	CS	0
LINECNT	L	57
NAME	N	MAIN
OPT	O	2

Simple Option Descriptions

BCD or EBCDIC

The BCD option indicates that the source module lines have been coded in Binary Coded Decimal (as on the 026 keypunch). The standard for the System/370 is Extended Binary Coded Decimal Interchange Code (EBCDIC). Most terminals and the 029 keypunch produce EBCDIC code. If the BCD option is specified, statement numbers passed as arguments must be coded as "$n" and "$" must not be used as an alphabetic character. Normally, statement numbers would be coded "&n", so that "$" would be a legitimate alphabetic character. The default is EBCDIC.

The compiler does not support BCD characters in either literal data or as print control characters; such characters are treated as EBCDIC. Consequently, for example, a BCD "+" used as a carriage-control character will not cause printing to continue on the same line. Programs keypunched in BCD should be carefully scanned for possible errors relating to print control characters and literal data.

COMMENT

The NOCOMMENT option inhibits the listing of comment statements. The default is COMMENT.

DECK

The DECK option specifies that object modules are to be produced by the compiler and that an identification field is to be generated for positions 73-80 of each object module record. The identification field consists of the first four characters of the module name, while the last four are sequentially numbered 0001, 0002, ... The default is DECK.

EJECT

The NOEJECT option causes the FORTRAN-H compiler to change all page ejects to triple spaces. This may be useful for some terminals. The default is EJECT.

ERR

The ERR option requests that source diagnostics be printed on SERCOM. The default is ERR.

ID

The ID option generates internal statement numbers (ISN) following external function references in the object code produced by the compiler. (An internal statement number is that number which would be attached to the statement if each executable statement were numbered sequentially from the beginning of the program.) In the generated code, each BALR (branch) to an external program is followed by a four-byte no-operation with an address field equal to the ISN of the source statement containing the external reference. For example, for a call from statement 1000 the four bytes would appear in a hexadecimal dump as 470003E8 (3E8 of base 16 = 1000 of base 10). The default is NOID.

LIST

The LIST option includes a pseudo-assembly-language format listing of the generated object module in the output listings. This listing consists of several columns that contain the hexadecimal representation of each instruction, a symbolic representation of

each instruction, the first instruction of each block corresponding to each statement label in the program, and a symbolic interpretation of any variable referenced by each instruction. Unless the user understands some machine code and can read hexadecimal dumps, this listing is generally not useful and is in all cases expensive to obtain. The use of the SCAN option overrides the LIST option as there is no object module produced when that option is employed. The default is NOLIST.

LOAD

The LOAD option specifies that object modules are to be produced by the compiler and included in the object module output. LOAD has no effect if used with the SCAN option. If the DECK option is specified, each record will be 80 characters long; otherwise, the object records will be 72 characters long. The default is NOLOAD.

Object module lines contain a 12-2-9 punch in the first column and the characters ESD, TXT, RLD, or END in columns 2-4. The compiler generates four types of ESD items: type 0 items contain the module name, entry point, and module length; type 1 items contain the entry point names corresponding to ENTRY statements; type 2 items contain the external references made in CALL or EXTERNAL statements and implicit or explicit function references; and type 5 items contain the names for each COMMON block. The FORTRAN-H compiler does not generate type 3 or type 4 ESD items. The TXT records contain user- and compiler-generated constants, translated FORMAT statements, and the generated machine instructions. The information contained in the RLD records is used by the loader to complete external references. External references are resolved by adjusting the constant pointed to by the address in the RLD item by the address of the appropriate external symbol contained in one of the type 2 ESD items. The END record for each module is described below:

Columns	Contents
1	12-2-9
2-4	END
37-39	FTN
41-48	Module name
49-56	Date as MM-DD-YY
57-64	Time as HH:MM.SS
65-68	Number of warning errors
69-72	Number of serious errors

If the DECK option is specified, then each object module record is suffixed with the first four characters of the module name in columns 73-76 and the line sequence in columns 77-80. The default is NOLOAD.

Note that when the SCAN option is specified, no punched output will be generated regardless of the LOAD or DECK option specifications.

MAP

The MAP option specifies that a storage map is to be included in the output listings. The map consists of tables containing variable names and locations for COMMON, EQUIVALENCE, NAMELIST, scalar, and array variables, subprograms referenced, and FORMAT statements. In addition, the compiler produces a label map. The map includes:

(1) the statement number of each source label,
(2) the relative address assigned to each label, and
(3) the symbol 'NR' next to each source label that is not referenced.

The default is MAP.

PRINT

The NOPRINT option causes the FORTRAN-H compiler to suppress all printed output including cross-reference and structured listings. The default is PRINT.

SCAN

The SCAN option causes the compiler to scan the source module for syntax and compilation errors. All appropriate error diagnostics are generated for each source module, but no object code is generated and consequently neither the object listing nor the program size can be included in the output listings. In addition, no object module output is produced. Use of the SCAN option will decrease the cost of scanning the source program for errors. The default is NOSCAN.

SOURCE

The SOURCE option produces a listing of the source deck in the output listings. If NOSOURCE is specified and source errors are found, the source statement in error and the diagnostic message are still included in the output listings. Note that even if NOSOURCE is specified, each module will produce at least 9 lines of messages. The default is SOURCE.

STRUC

The STRUC option specifies that a structured source listing is to be produced. This listing indicates the loop structure and the logical continuity of the source program. The STRUC option is effective only if OPT=2 is also specified. The default is NOSTRUC. For a complete description of this listing, see the IBM System/360 Operating System FORTRAN IV (G and H) Programmer's Guide, form GC28-6817.

TEST

The TEST option generates SYM records in the object module. This facilitates execution-time debugging with the Symbolic Debugging System (SDS). See the section "Introduction to Debug Mode for FORTRAN" for further information. The default is NOTEST.

XL

The XL option specifies that the FORTRAN-H compiler extended language features are permitted in the source deck. For details of these features, see the IBM publication IBM System/360 Operating System FORTRAN IV (H) Compiler Program Logic Manual, form GY28-6642. The NOXL option specifies that the extended language features are not permitted. Note that this is not the same as the FORTRAN-H Extended compiler (an IBM program product), which is not available at the Computing Center.

XREF

The XREF option produces a cross-reference listing of variables and labels in the output listings. The variable names are listed in alphabetical order according to length. (Variable names of one character appear first in the listing.) The labels are listed in ascending sequence along with the internal statement number of the statement in which the label is defined. For both variable names and labels the listing also contains the internal statement number of each statement in which the variable or label is used.

Assignment Option Descriptions

CALIGN=n
CSHIFT=n

By specifying the CALIGN=n or CSHIFT=n options (where $1 \leq n \leq 72$), the body of comment lines may be positioned to start in column "n". The comment lines are denoted by column 1 containing the comment symbol "C". CALIGN aligns the first nonblank character after the initial "C" at column "n". CSHIFT shifts the entire comment (including the "C") to column "n", thus preserving the indentation of structured comments. For both CALIGN and CSHIFT, a "C" is printed at column 1 in the listing. Blank comments are not shifted. If "n" is 72, the comments will be printed in the normally blank area to the right of the source statement listing. If truncation would occur, the comment is continued on the next line. Standard output is printed with CALIGN=0 and CSHIFT=0 (the defaults). CALIGN=1 is treated as if CALIGN=2 were specified. If "n" is greater than 72, 72 will be used. CALIGN and CSHIFT are "coupled" options, i.e., the last one specified overrides all

previous occurrences of either. These options are available only with the FORTRAN-H compiler.

LINECNT=n

The LINECNT assignment option specifies the number of lines per page for the output listings. "n" must be an integer in the range (3, 32767). This may be used to control the spacing of the page headers, etc., as they are only produced at the beginning of each set of lines. The default is 57 lines per page.

NAME=xxxx

The NAME assignment option specifies the name to be used as the module name for all main programs compiled. Subprogram module names are always the name given in the FUNCTION or SUBROUTINE statements. The module name appears in the page headers, LIB records, the ESD type 0 items, the END record, and the object code at relative address 000005. The name "xxxx" may be from 1 to 8 characters in length. The default is MAIN.

OPT=n

The OPT assignment option specifies the optimization level to be used in the compilation of the program. The OPT=0 option indicates that the compiler uses no optimizing techniques in producing an object module. The OPT=1 option indicates that the compiler treats each source module as a single program loop and optimizes the loop with regard to register allocation and branching. The OPT=2 option indicates that the compiler treats each source module as a collection of program loops and optimizes each loop with regard to register allocation, branching, common expression elimination, and replacement of redundant computation. For a more detailed description of the optimization procedures, see the section "FORTRAN-H Optimization Facilities."

USE OF FORTRAN-H AS A SUBROUTINE

The FORTRAN-H compiler may be called as a subroutine. The compiler must be concatenated to the object (calling) program on the $RUN command, e.g.,

```
$RUN object+*FORTRANH [logical I/O units]
```

The calling sequence is

```
CALL FTNH(options,reader,printer,punch,break,errrtn,&rc4,...,&rc16)
```

All of the arguments except the first are optional. The calling sequence must conform to the FORTRAN standard described in MTS Volume 3,

System Subroutine Descriptions. Specifically, since most of the arguments are not required, the high-order byte of the last argument address must contain X'80'. Although only the first argument is required, in order to specify one of the optional arguments, all of the preceding arguments must also be specified, i.e., if one wanted to specify the printer, he would also have to specify the reader, e.g.,

```
CALL FTNH(options,reader,printer,&rc4,...,&rc16)
```

All of the optional arguments are subroutine names and must be declared in an EXTERNAL statement in a FORTRAN program. For assembly language users, the optional arguments are passed by placing a pointer to the appropriate V-type address constant in the parameter list.

The arguments to FTNH are described below.

Options

If the parameter field length is negative, then the options parameter is the address of an eight (8) halfword vector formatted as follows: four halfwords containing the eight-character default main program name, left-justified with trailing blanks, two halfwords containing the option bits, one halfword containing the number of lines per page, and one halfword containing the memory block size. Within the first option bit halfword, the parameter is enabled if the corresponding bit is set; otherwise, it is disabled. The bit assignments are as follows:

Bit	Hex	Function	Default	
0	80000000	SCAN	0	
1	40000000	EJECT	1	
2	20000000	PRINT	1	
3	10000000	COMMENT	1	
4	08000000	TEST	1	
5	04000000	ERR	1	
6	02000000	XL	0	
7	01000000	XREF	0	
8	00800000	ID	1	
9	00400000	STRUC	0	
10	00200000	MAP	1	
11	00100000	LOAD	0	
12	00080000	DECK	1	
13	00040000	LIST	0	
14	00020000	BCD	0	
15	00010000	SOURCE	1	
18	00002000	CSHIFT	0	
19	00001000	CALIGN	0	
31	00000001	EXTEN	0	(Future option, should be 0)

The remaining bits should be 0 (zero). Bits 18 and 19 (CSHIFT and CALIGN) should not both be 1. If CSHIFT or CALIGN is 1, the value of the alignment is passed in a single fullword appended to the end of the OPTION area, making the region 10 halfwords instead of 8.

The following example from a BLOCK DATA program illustrates how the default options may be set up to be passed in the direct form:

```
COMMON /OPTION/ NAME,OPT,LINE,SIZE,OPTIM
REAL*8 NAME/'MAIN'/
INTEGER*2 OPT(2)/Z0129,0/,LINE/57/,SIZE/0/
INTEGER*4 OPTIM/2/
```

Note that when the options are passed directly that they completely replace the default options, rather than modify them. It is not possible to simply change some of the option bits.

Reader

This argument is the name of an external function to be called to obtain the source input lines. If this optional argument is not given, the default value is SCARDS. The parameter list is the same as that described for the MTS READ subroutine in MTS Volume 3. The unit described for the READ subroutine is set to logical I/O unit 0. Although there are several parameters in the READ subroutine calling sequence, the user need only be concerned with the first. FTNH passes the address of an 80-character, preblanked buffer, and expects that the source will be placed in the buffer. For example:

```
      SUBROUTINE READER(BUFFER,*)
      REAL*8 BUFFER(10)
         .
         .
      RETURN
  100 RETURN 1
      END
```

Any nonzero return code passed by the reader routine to the compiler will be interpreted as an end-of-file. On an end-of-file indication the buffer contents are ignored. Any attempt to place more than 80 characters in the buffer will result in a fatal compiler error. If fewer than 80 characters are placed in the buffer, then the blanks that are placed in the buffer prior to the call may be used to pad the input line. The set of source modules to be compiled must be in standard IBM format and terminated by an end-of-file condition, i.e., a nonzero return code from the routine.

The reader routine may return an MTS line number in the appropriate parameter. If this is done, *FORTRANH will print the line number in the listing. If the parameter is unchanged, no line number will be printed.

Printer

This argument is the name of an external function to be called to dispose of the output listing lines produced by the compiler. If the optional argument is not given, a default value of SPRINT is used. The printer parameter list corresponds to that described for the WRITE subroutine in MTS Volume 3. The unit described in that parameter list is set to MTS logical I/O unit 1. Although there are several arguments, the programmer need only be concerned with the first, the 120-character output line.

The printer output line consists of 120-character output lines with the logical carriage-control characters 0, 1, or blank. The amount and type of information is controlled by the options SOURCE, MAP, and LIST. Each source module will generally produce at least two lines, a page heading containing the module name, date, and time, and a line giving the total memory storage requirements of the program in bytes. Any diagnostics are included regardless of the options requested.

See the descriptions of the SOURCE, MAP, and LIST parameters for a description of the output that is produced for each option.

Punch

This argument is the name of an external function which will be called to dispose of any object module lines produced by the compiler. If the argument is not given, a default value of SPUNCH is supplied. It should be noted that unless one is running in batch, with a nonzero card estimate, SPUNCH is not defaulted. When defaulted, SPUNCH defaults to *PUNCH*. The parameter list for the call to the punch routine is the same as that for the system subroutine WRITE as described in MTS Volume 3. Unit corresponds to logical I/O unit 2. Of the arguments in the call, only the first two are of importance to a user-supplied routine. These arguments are the buffer containing the object module line and a halfword integer (INTEGER*2) output line length, respectively. The object module lines will be 80 character if the DECK parameter is specified; otherwise, they will be 72.

Break

This argument is the name of an external function to be called after each source module is compiled and before the next compilation has begun. If this optional argument is not given, the compiler simply proceeds to the compilation of the next source module.

If the BREAK routine gives a nonzero return code to the compiler, it will immediately halt and return to its calling program with the appropriate return code.

Errrtn

The optional parameter "errrtn" is a subroutine called by FORTRAN H to process error messages. All errors will be sent to the PRINT subroutine regardless of the presence of this parameter. If no errors are encountered, a message to that effect is passed to the "errrtn" subroutine. The calling sequence is the same as for the other output subroutines. If the ERR option is reset, "errrtn" will not be called.

Return Codes

The compiler returns in the normal FORTRAN manner (see a description of FORTRAN calling sequences in MTS Volume 3). In addition, the return code is also placed in general register 0 (zero), so that FTNG may be declared an INTEGER*4 function, and as such, will assume the values 0, 4,...,16. The return codes and explanations follow.

Return Code	Meaning
0	All compilations have been completed and no errors were found.
4	All compilations have been completed and only diagnostics given have severity level 4.
8	All compilations have been completed and at least one severity level 8 diagnostic was given.
12	Not used.
16	Compiler malfunction. Try compiling the source module again and if the error persists, consult with a Computing Center counselor.

USE OF THE FORTRAN-H COMPILER AS A STAND-ALONE LANGUAGE PROCESSOR

As indicated earlier, it is not recommended that the FORTRAN-H compiler be used as a stand-alone language processor. However, use of FORTRAN-H in this manner will provide an efficient batch-oriented interface to the compiler. Conversational users will find this version of the compiler inconvenient to use and should refer to the section "*FTN Interface" in this volume. The following deck structure would be sufficient to compile a set of FORTRAN-H programs punched on cards according to the standard IBM format.

```
$SIGNON ccid 'name'
password
$RUN *FORTRANH SPUNCH=-LOAD PAR=LIST
        .
    (source program)
        .
$ENDFILE
$RUN -LOAD
$SIGNOFF
```

The characters "ccid" are the user's four-character Computing Center signon ID. The second card contains the user's password starting in column 1. The LIST option is specified to produce an object listing. Note that the options are passed to the compiler via the PAR field on the $RUN command.

FORTRAN-H OPTIMIZATION FACILITIES

This section contains information relating to the use of the FORTRAN-H compiler optimization facilities. It is reprinted with permission from the IBM publication, IBM System/360 Operating System FORTRAN IV (G and H) Programmer's Guide, form GC28-6817.

Program Optimization

Facilities are available in the FORTRAN IV (H) compiler that enable a programmer to optimize execution speed and to reduce the size of the object module. There are three levels of optimization available: 0, 1, and 2. Optimization level 0 provides no optimization, optimization level 1 provides some, and optimization level 2 provides the most optimization.

When using OPT=1, the entire program is treated as a loop, while individual sections of coding, headed and terminated by labeled statements, are blocks. The object code is improved by:

(1) Improving local register assignment. (Variables that are defined and used in a block are retained (if possible) in registers during the processing of the block. Time is saved because the number of load and store instructions are reduced.)

(2) Retaining the most active base addresses and variables in registers across the whole program. (Retention in registers saves time because the number of load instructions is reduced.)

(3) Improving branching by the use of RX branch instructions. (An RX branch instruction saves a load instruction and reduces the number of required address constants.)

When using OPT=2, the loop structure and data flow of the program are analyzed. The object code is improved over OPT=1 by:

(1) Assigning registers across a loop to the most active variables, constants, and base addresses within the loop.

(2) Moving outside the loop many computations which need not be calculated within the loop.

(3) Recognizing and replacing redundant computations.

(4) Replacing (if possible) multiplication of induction variables by addition of those variables.

(5) Deleting (if possible) references to some variables.

(6) Using (where possible) the BXLE instruction for loop termination. (The BXLE instruction is the fastest conditional branch; time and space are saved.)

The variables that FORTRAN H considers "optimizable" are marked with an asterisk "*" in the cross-reference listing.

Programming Considerations Using the Optimizer

In general, the specification of OPT=1 or OPT=2 causes compilation time to increase. However, the object code produced is more concise and yields shorter execution times.

The object module logic, when optimized, is identical to the unoptimized logic, except in the following cases:

(1) If the list of statement numbers in an Assigned GO TO statement is incomplete, errors, which were not present in the unoptimized code, may arise in the optimized code.

(2) With OPT=2, the computational reordering done may produce a different execution time behavior than unoptimized code. Consider the following example:

```
          DO 11 I=1,10
          DO 12 J=1,10
          IF (B(I).LT.0.) GOTO 11
       12 C(J)=SQRT(B(I))
       11 CONTINUE
```

The square root computation will be moved backward outside the inner loop and hence will occur before the less-than-zero test. This will result in a message if B(I) is negative. A rearrangement of the program which could avoid this situation can be constructed:

```
          DO 11 I=1,10
          IF (B(I).LT.0.) GOTO 11
          DO 12 J=1,10
       12 C(J)=SQRT(B(I))
       11 CONTINUE
```

(3) If a programmer defines a subprogram with the same name as a FORTRAN-supplied subprogram (e.g., SIN, ATAN, etc.), errors could be introduced during optimization. If the subprogram stores into its arguments, refers to COMMON, performs I/O, or remembers its own variables from one execution to another, the name of the subprogram must be specified in an EXTERNAL statement to allow the program to be optimized without error.

(4) In the statements

```
          COMMON X,Y1(10),W,Z
          EQUIVALENCE (Y1,Y2)
          DIMENSION Y2(12)
```

there is an implied equivalence of Y2(11) and W and Y2(12) and Z. If the optimization feature is not used, and the statements

```
          W=Q
          A=Y2(I)   (where I=11)
```

are executed, the value of Q is assigned to A. However, if OPT=2 is used, and the statements

```
          W=Q
          A=Y2(I)   (where I=11)
```

are executed, there is no guarantee that the value of Q is assigned to A.

(5) When a subprogram is called at one entry point for initialization of reference-by-name arguments, and at another entry point

for subsequent computation, certain argument values may not be transmitted. This applies to either arguments of the second call or any argument values redefined between calls and not explicitly defined in COMMON.

In the following example the incremented value for I may not be transmitted to the subprogram due to the loop initialization optimization. This is because the value of I may be contained in a register throughout the loop and not stored into the memory location for I until the loop is exited.

```
      CALL INIT(I)            SUBROUTINE INIT(/J/)
          .                       .
          .                       .
          .                       .
      I = 0                   ENTRY COMP
   10 CALL COMP
      I = I+1
          .
          .
          .
      GOTO 10
```

(6) With OPT=2, variables in named COMMON arrays may not be stored on exit from a FORTRAN main program if these variables have not been used in an I/O statement in that main program, or if there is no subroutine call following the definition of these variables.

(7) With OPT=2, implied DO variables may not be stored if an END= transfer was made out of a READ statement.

Use of Loops

The FORTRAN-H compiler treats a DO-loop as an actual loop. Additionally, the compiler may treat any other sequence of statements that appear to be executed iteratively as a loop. However, the compiler may not treat as loops other sequences of statements which the programmer perceives as loops.

If a programmer writes a loop which is preceded by an IF statement, a conditional GO TO statement, or READ statement with END or ERR options, the loop is not identified and efficiency is lost. A CONTINUE statement at the end of the range of a DO also obscures a loop (other than a DO loop) that follows the CONTINUE without intervening initialization. The insertion of a labeled CONTINUE statement or any other suitable rearrangement allows the loop to be recognized.

The movement of computations from inside a loop to the initialization coding is done on the assumption that every statement in the loop is

executed more frequently than the initialization coding. Occasionally, this assumption fails and computations are moved to a position where they are computed more often. One way to prevent such a move is to make a subprogram of the coding (statements and computations) that is executed less frequently within a loop than it would be in the initialization coding.

The recognition of loops may also be obscured when the programmer knows that some paths through the program cannot occur; for example,

```
 10 IF (L) GOTO 200
 20 I=1
 30 ASSIGN 40 TO J
    GOTO 100
 40 I=I + 1
 50 IF (I.LE.N) GOTO 30
      .
      .
      .
100 B(I) = FUNCT(I)
110 GOTO J, (40, 220)
200 ASSIGN 220 TO J
210 GOTO 100
220 CONTINUE
```

From the programmer's point of view, the statements 30 to 50 comprise a loop which is initialized by statement 20. The loop causes an internal subprogram consisting of statements 100 and 110 to be executed. From the compiler's point of view, it appears possible to execute statements in the order 10, 200, 210, 100, 110, 40, 50, 30. The compiler does not recognize the loop, because it appears possible to enter it without passing through the initialization coding in statement 20. A loop can be obscured by the computed GO TO, because the compiler always assumes that one of the possible branches is to the succeeding statement, even though the programmer knows that such a branch is impossible. A loop can also be obscured by a call to the EXIT routine, because the compiler assumes there is a path from such a statement to the next.

Movement of Code into Initialization of a Loop

Where it is logically possible to do so with OPT=2, the optimizer moves computations from inside the loop to the outside. This movement permits a programmer to do more straightforward coding without penalty in object code efficiency.

If an expression is evaluated inside a loop and all the variables in the expression are unchanged within the loop, the computation is generally moved outside the loop into the coding sequence which initializes the loop. Even if the constant expression is part of a

larger expression, this constant expression may still be recognized and moved. However, the movement depends on how the larger expression is written. The table below gives examples of expressions and the constant parts which are recognized and moved.

Expression where C1, C2,... are constant in the loop	Constant expression recognized and moved
C1 + C2 * C3/SIN (C4)	C1 + C2 * C3/SIN (C4)
C1 + C2 * C3 + B1	C1 + C2 * C3
C1 + B1 + C2 * C3	C2 * C3
B1 + C1 + C2 * C3	C2 * C3
C1 + B1 + B2 + C2 * C3	C2 * C3
C1 * C2/B1	C1 * C2

Common Expression Elimination

With OPT=2, if an expression occurs twice in such a way that:

(1) any path starting at an entry to the program always passes through the first occurrence of the expression to reach the second occurrence (and any subsequent occurrence), and

(2) any evaluation of the second (third, fourth, etc.) expression produces a result identical to the most recent evaluation of the first expression, then the value of the first expression is saved (generally) and used instead of the value of the second (third, fourth, etc.) expression.

In statements such as

```
A = B + C + D
E = C + D
```

the common expression C + D is not recognized, because the first expression is computed as (B + C) + D.

Induction Variable Optimization

In a loop with OPT=2, an induction variable is a variable that is only incremented by a constant or by a variable whose value is constant in the loop.

When an induction variable is multiplied by a constant in the loop, the optimizer may replace the multiplication with an addition by

introducing a new induction variable into the loop. This new induction variable may make it possible to delete all references to the original induction variable. This deletion is likely to occur if the original induction variable is used only as a subscript within the loop, and the value of the subscript is not used on exit from the loop.

Register Allocation

Some variables are assigned to a register on entry to a loop and retained in the register through part or all of the loop to avoid loading and storing the variable in the loop. Within the loop, the variable is modified only in the assigned register, the value of the variable in storage is not changed. If necessary, the latest value of the variable is stored after exit from the loop.

The value in general register 13, which points to the start of a register save area, remains constant during execution of a subprogram. This register is used to refer to data, and possibly to branch within the program. The value in general register 12 remains constant and is used to branch within the program, and possibly is used to refer to data.

General registers 14 and 15 are used for base addresses and index values on a strictly local basis. Floating-point register 0 and general register 0 are used as locally assigned arithmetic accumulators. General register 1 is used in conjunction with general register 0 for fixed-point arithmetic operations, and to point to argument lists in subprogram linkages.

The remaining registers are used for accumulators, index values, base addresses, and high-speed storage (a register reference is faster than a main storage reference).

Because general registers 12 and 13 are not adequate to provide RX branching throughout a large program, general registers 11, 10, and 9 may be preempted for RX branching (only if the program exceeds 8K, 12K, and 16K bytes, respectively). (RR branches preceded by loads are required for branching to points beyond the first 16K bytes of the program and possibly to the last part of the program if it exceeds 8K, 12K, or 16K bytes by a small amount.)

COMMON Blocks

Because each COMMON block is independently relocatable, each requires at least one base address to refer to the variables in it. A sequence of coding that refers to a large number of COMMON blocks is slowed down by the need to load base addresses into general registers. Thus, if

three COMMON blocks can be combined into one block whose total size is less than 4096 bytes, one base address can serve to refer to all the variables. (Many register loads can be avoided.)

The order in which data are entered into a COMMON block may also affect the number of base addresses needed. For example, if an array of 5000 bytes is placed in a COMMON block and followed by 200 bytes of variables, two base addresses are needed: the beginning address of the first variable and the beginning address of the last differ by more than 4096 bytes. However, if the variables preceded the array, one base address would suffice.

EQUIVALENCE Statements

Optimization tends to be weakened by the occurrence of variables in EQUIVALENCE statements.

When an array appears in an EQUIVALENCE statement, a reference to one of its elements cannot be eliminated as a common expression, nor can the reference be moved out of a loop. However, the elimination and movement of subscript calculations used for making the reference is not affected.

If a variable is made equivalent only to another variable (not in COMMON) of the same type and length, optimization is not weakened. The net effect is that the compiler accepts the two names as alternate pointers to the same storage location. However, if a variable is made equivalent to another variable in any other way, all references to it are "immobilized": the references cannot be eliminated, moved, confined to registers, or altered in any way.

Multidimensional Arrays

In general, references to higher dimensional arrays are slower than references to lower dimensional arrays. Thus, a set of one-dimensional arrays is more efficient than a single two-dimensional array in any case where the two-dimensional array can be logically treated as a set of one-dimensional arrays.

Constants occurring in subscript expressions are accounted for at compile time and have no effect at execution time.

Program Structure

If a large number of variables are to be passed among calling and called programs, some of the variables should be placed in the COMMON area. For example, in the main program and subroutine EXAMPL

```
DIMENSION E(20),I(15)
READ(10) A,B,C
CALL EXAMPL(A,B,C,D,E,F,I)
   .
   .
   .
END

SUBROUTINE EXAMPL(X,Y,Z,P,Q,R,J)
DIMENSION Q(20),J(15)
   .
   .
   .
RETURN
END
```

time and storage are wasted by allocating storage for variables in both the main program and subprogram and by the subsequent instructions required to transfer variables from one program to another.

The two programs should be written using a COMMON area, as follows:

```
COMMON A,B,C,D,E(20),F,I(15)
READ(10) A,B,C
CALL EXAMPL
   .
   .
   .
END

SUBROUTINE EXAMPL
COMMON X,Y,Z,P,Q(20),R,J(15)
   .
   .
   .
RETURN
END
```

Storage is allocated for variables in COMMON only once and fewer instructions are needed to cross-reference the variables between programs.

To reduce compilation time for equivalence groups, the entries in the EQUIVALENCE statement should be specified in descending order according to offset. For example, the statement

```
      EQUIVALENCE (ARR1(10,10),ARR2(5,5),ARR3(1,1),VAR1)
```

compiles faster than the statement

```
      EQUIVALENCE (VAR1,ARR3(1,1),ARR2(5,5),ARR1(10,10))
```

To reduce compilation time and save internal table space, equivalence groups should be combined, if possible. For example, the statement

```
      EQUIVALENCE (ARR1(10,10),ARR2(5,5),VAR1)
```

compiles faster and uses less internal table space than the statement

```
      EQUIVALENCE (ARR1(10,10),VAR1),(ARR2(5,5),VAR1)
```

Logical IF Statements

A statement such as

```
      IF (A.LT.B .OR. C.GT.F(X) .OR. .NOT.L) GOTO 10
```

is compiled as though it were written

```
      IF (A.LT.B) GOTO 10
      IF (C.GT.F(X)) GOTO 10
      IF (.NOT.L) GOTO 10
```

Thus, if A.LT.B is found to be true, the remainder of the logical expression is not evaluated.

Similarly, a statement such as

```
      IF (D.NE.7.0 .AND. E.GE.G) I=J
```

is compiled as

```
      IF (D.EQ.7.0) GOTO 20
      IF (E.LT.G) GOTO 20
      I=J
   20 CONTINUE
```

The order in which a programmer writes logical expressions in an IF statement affects the speed of execution.

If A is true more often than B, then write A .OR. B rather than B .OR. A; and write B .AND. A rather than A .AND. B.

If any of the following occur in a logical expression:

(1) a mixture of both .AND. and .OR. operators, or
(2) a .NOT. operator followed by a parenthesized expression

the entire logical expression must be evaluated and efficiency is lost.

Branching

The statement

```
        IF (A.GT.B) GOTO 20
```

gives equivalent or better code than

```
          IF (A-B) 10,10,20
       10 CONTINUE
```

The assigned GOTO is the fastest conditional branch.

The computed GOTO should be avoided unless four or more statement labels occur within the parentheses.

The statement

```
        IF (I-2) 20,30,40
```

is significantly faster than

```
          GOTO (20,30,40), I
```

FORTRAN-H SOURCE MODULE ERROR/WARNING MESSAGES

At the end of each compilation, the FORTRAN-H compiler prints a set of statistics in a format similar to the following:

```
                      **** ERRORS FOR SUBR ****

  0064    162.000       TEST = .FALSE

  IEK224I (08)          THE EXPRESSION HAS AN INVALID DOUBLE DELIMITER.

  IEK610I (04)    1006  THE STATEMENT NUMBER OR GENERATED LABEL IS
                        UNREACHABLE.

  *OPTIONS IN EFFECT*    NAME=  MAIN,OPT=02,LINECNT=58,

  *OPTIONS IN EFFECT*    SOURCE,EBCDIC,NOLIST,DECK,NOLOAD,MAP,NOSTRUC,
                         NOID,XREF
```

```
*STATISTICS*          SOURCE STATEMENTS =    8 ,PROGRAM SIZE =    366

*STATISTICS*          2 DIAGNOSTICS GENERATED, HIGHEST SEVERITY CODE
                      IS 8

******END OF COMPILATION******

COMPILER STATISTICS:  ELAPSED TIME     10.790 SEC.
                      CPU TIME     .660 SEC.
```

The above example indicates how the error messages are denoted. If there are no compilation errors, the message:

```
*STATISTICS* NO DIAGNOSTICS GENERATED
```

appears in place of the error and diagnostic messages. The letters ISN in the error messages refer to the internal statement number of the statement in error.

The following is a list of error messages with a brief explanation of each. The messages are sequenced in ascending numerical order according to the number "xxx" in IEKxxxI. In addition to the message at the end of the compilation, each statement flagged with a serious error is followed by the message

```
ERROR DETECTED - SCAN POINTER = x.
```

where "x" represents the position of the character pointed to by the compiler's internal scan pointer when the error was detected. FORTRAN keywords and/or meaningless blanks are ignored in determining the position of the pointer. If the statement is found to be invalid during the compiler's classification process, the value of "x" is set to one. Error messages which are self-explanatory do not have any additional comment.

IEK001I THE NUMBER OF ENTRIES IN THE ERROR TABLE HAS EXCEEDED THE MAXIMUM.

Too many statements have been found to contain errors. Correct those statements found to be in error and resubmit the program. Severity Level: 8.

IEK002I THE DO LOOPS ARE INCORRECTLY NESTED.

This diagnostic is generated if the statements in the range of an inner DO LOOP are not contained in the range of the outer DO. This message will also appear if the extended range of a DO statement contains another DO with an extended range and both DOs are in the same main program or subroutine. Severity Level: 8.

IEK003I THE EXPRESSION HAS AN INVALID LOGICAL OPERATOR.

Look at all the logical operators for misspelling or improper placement. Severity Level: 8.

IEK005I THE STATEMENT HAS AN INVALID USE OF PARENTHESES.

This diagnostic will occur if there are mismatched parentheses, or if parentheses appear where they are not allowed. Severity Level: 8.

IEK006I THE STATEMENT HAS AN INVALID LABEL.

This diagnostic will appear if the user attempts to label a statement in an invalid manner. For example, this error message will appear when the user attempts to branch to the label of a format statement. Severity Level: 8.

IEK007I THE EXPRESSION HAS AN INVALID DOUBLE DELIMITER.

Look at the use of all commas, parentheses, primes, and blanks in the indicated expression. Correct any illegal delimiters. Severity Level: 8.

IEK008I THE EXPRESSION HAS A CONSTANT WHICH IS GREATER THAN THE ALLOWABLE MAGNITUDE.

The constants must be within the following ranges:

Type	Range
INTEGER	-2147483647, 2147483647
REAL	.53E-78, .72E+76
REAL*8	.53E-78, .72E+76

Severity Level: 8.

IEK009I THE EXPRESSION HAS A NONNUMERIC CHARACTER IN A NUMERIC CONSTANT.

Severity Level: 8.

IEK010I THE EXPRESSION HAS A CONSTANT WITH AN INVALID EXPONENT.

This diagnostic will result if the base and exponent are invalid combinations of operand types. For example, a real number used in an exponent: 1.06E+.05. Severity Level: 8.

IEK011I THE ARITHMETIC OR LOGICAL EXPRESSION USES AN EXTERNAL FUNCTION NAME AS A VARIABLE NAME.

To correct this problem, make sure that each name in an EXTERNAL statement is not also used as a variable in an expression. Use of the MAP option can help determine how each name is perceived. Severity Level: 8.

IEK012I THE EXPRESSION HAS A COMPLEX CONSTANT WHICH IS NOT COMPOSED OF REAL CONSTANTS.

Both parts of the complex constant, real and imaginary, must be real numbers. The decimal point must be explicitly specified in both parts. Severity Level: 8.

IEK013I AN INVALID CHARACTER IS USED AS A DELIMITER.

Check use of primes, semicolons, etc. Severity Level: 8.

IEK014I THE STATEMENT HAS AN INVALID NONINTEGER CONSTANT.

Severity Level: 8.

IEK015I THE ARITHMETIC OR LOGICAL EXPRESSION USES A VARIABLE NAME AS AN EXTERNAL FUNCTION NAME.

This is the opposite of IEK011I above. Look at the use of all variables in the expression. Severity Level: 8.

IEK016I THE GO TO STATEMENT HAS AN INVALID DELIMITER.

Check the format of the statement in use; blanks, commas, and parentheses are the only valid delimiters and their usage depends on the type of GO TO. Severity Level: 8.

IEK017I THE ASSIGNED OR COMPUTED GO TO HAS AN INVALID ELEMENT IN ITS STATEMENT NUMBER LIST.

The statement numbers list must contain only labels for executable statements. If an assigned GO TO is in question, check the ASSIGN statement as well. Severity Level: 8.

IEK019I THE ASSIGNED GO TO HAS THE OPENING PARENTHESIS MIS-PLACED OR MISSING.

Severity Level: 8.

IEK020I THE ASSIGNED GO TO HAS AN INVALID DELIMITER FOLLOWING THE ASSIGNED VARIABLE.

The delimiter must be a comma. Severity Level: 8.

IEK021I THE COMPUTED GO TO HAS AN INVALID COMPUTED VARIABLE.

The delimiter must be a comma. Severity Level: 8.

IEK022I THE VARIABLE IN THE ASSIGNED GO TO STATEMENT IS NOT INTEGRAL.

If the variable appears to be integral, check both type and IMPLICIT DECLARATIONS. Severity Level: 8.

IEK023I THE DEFINE FILE STATEMENT HAS AN INVALID DATA SET REFERENCE NUMBER.

The data set reference number must be an unsigned integer constant. Severity Level: 8.

IEK024I THE DEFINE FILE STATEMENT HAS AN INVALID DELIMITER.

Check the format of the statement, noting especially the placement of commas and parentheses. Severity Level: 8.

IEK025I THE DEFINE FILE STATEMENT HAS AN INVALID INTEGER CONSTANT AS THE RECORD NUMBER OR SIZE.

Severity Level: 8.

IEK026I THE DEFINE FILE STATEMENT HAS INVALID FORMAT CONTROL CHARACTER.

The valid format control characters are L, E, and U. Severity Level: 8.

IEK027I THE ASSIGN STATEMENT HAS AN INVALID INTEGER VARIABLE.

The integer variable must not be subscripted. Severity Level: 8.

IEK028I THE ASSIGN STATEMENT HAS AN INVALID DELIMITER.

Check the format of the ASSIGN statement, noting the comma and parenthesis locations. Severity Level: 8.

IEK030I THE DO STATEMENT HAS AN INVALID END OF RANGE STATEMENT NUMBER.

This diagnostic will occur if the end of range statement number labels a nonexecutable statement. Severity Level: 8.

IEK031I THE DO STATEMENT OR IMPLIED DO HAS AN INVALID INITIAL VALUE.

The initial value of a DO must be an unsigned integer constant greater than zero, or an unsigned nonsubscripted integer variable greater than zero. Severity Level: 8.

IEK034I THE ASSIGNMENT STATEMENT BEGINS WITH A NONVARIABLE.

Check other uses of the name on the left side of the equal sign. Use the MAP option to determine the nature of the unknown. Severity Level: 8.

IEK035I THE NUMBER OF CONTINUATION CARDS EXCEEDS THE COMPILER LIMIT.

There should not be more than 19 continuation cards in a row. Severity Level: 8.

IEK036I THE STATEMENT CONTAINS INVALID SYNTAX. THE STATEMENT CANNOT BE CLASSIFIED.

This is a catchall diagnostic for anything that the compiler cannot identify as a legitimate statement type. Look for mispunches, misspellings, etc. Severity Level: 8.

IEK039I THE DEFINE FILE STATEMENT HAS AN INVALID ASSOCIATED VARIABLE.

The associated variable must be integral and nonsubscripted. In addition, it must not appear in an I/O list for a READ or WRITE associated with the file of the DEFINE FILE statement. Severity Level: 8.

IEK040I IT IS ILLEGAL TO HAVE A & STATEMENT NUMBER PARAMETER OUTSIDE A CALL STATEMENT.

Look at the statement and see what is meant. Delete the & statement number. Severity Level: 8.

IEK044I ONLY THE CALL, FORMAT, OR DATA STATEMENTS MAY HAVE LITERAL FIELDS.

This diagnostic is generated if an attempt is made to use a literal field outside of one of the above statements. If an *WATFIV deck is being compiled, check the WRITE statements. Severity Level: 8.

IEK045I THE EXPRESSION HAS A LITERAL WHICH IS MISSING A DELIMITER.

Literal delimiters, like parentheses, must be paired. Severity Level: 8.

IEK047I THE LITERAL HAS MORE THAN 255 CHARACTERS IN IT.

This diagnostic could be generated by a missing delimiter. Insert a delimiter (usually a prime) or shorten the expression to correct the problem. Severity Level: 8.

IEK050I THE ARITHMETIC IF HAS THE SYNTAX OF THE BRANCH LABELS INCORRECT.

There should be three executable statement numbers with commas following the first and second statement numbers. Severity Level: 8.

IEK052I THE EXPRESSION HAS AN INCORRECT PAIRING OF PARENTHESES OR QUOTES.

This diagnostic will be generated if an H format specification is larger than the data and encompasses a final parenthesis. It will occur if quotes within literals are not represented by two successive quotes. Severity Level: 8.

IEK053I THE STATEMENT HAS A MISPLACED EQUAL SIGN.

Severity Level: 8.

IEK056I THE FUNCTION STATEMENT MUST HAVE AT LEAST ONE ARGUMENT.

Severity Level: 8.

IEK057I THE STATEMENT HAS A NONVARIABLE SPECIFIED AS A SUBPROGRAM NAME.

Subprogram names must conform to the usual FORTRAN standards for variable names, i.e., 1-6 alphanumeric characters, beginning with an alphabetic character. Severity Level: 8.

IEK058I THE SUBPROGRAM STATEMENT HAS AN INVALID ARGUMENT.

One of the arguments in the subroutine definition is an invalid argument. The MAP option can be used to help determine the types of all variables in the statement. This diagnostic will be generated if an attempt is made to specify a constant in place of a dummy argument in the subprogram statement. Severity Level: 8.

IEK059I THE FUNCTION STATEMENT HAS AN INVALID LENGTH SPECIFICATION.

The length specification must not be given if the function is declared DOUBLE PRECISION. In all other cases the length specification must be one that is permissible for the declared type; e.g., INTEGER*4, INTEGER*2. Severity Level: 8.

IEK061I THE EQUIVALENCE STATEMENT CONTAINS A NONSUBSCRIPTED ARRAY ITEM. INCORRECT ADCONS MAY BE GENERATED.

Make sure that all array references in the associated equivalence statements are subscripted to prevent improper association. Severity Level: 4.

IEK062I THE EQUIVALENCE STATEMENT HAS AN ARRAY WITH AN INVALID NUMBER OF SUBSCRIPTS.

The number of subscripts for the array in the equivalence statement must be the same as that in the specification statement for the array. Severity Level: 8.

IEK064I THE NAMELIST STATEMENT HAS AN INVALID DELIMITER.

Check the format of the NAMELIST statement; it should contain only '/' and ',' as delimiters. Severity Level: 8.

IEK065I THE NAMELIST STATEMENT HAS A NAMELIST NAME NOT BEGINNING WITH AN ALPHABETIC CHARACTER.

Severity Level: 8.

IEK066I THE NAMELIST STATEMENT HAS A NONUNIQUE NAMELIST NAME.

The namelist name must be unique. It cannot be the same as a variable or array name, and cannot appear in more than one NAMELIST declaration. Severity Level: 8.

IEK067I THE NAMELIST STATEMENT HAS AN INVALID LIST ITEM.

The list items must be variables, array items, or array names. It is possible for a variable or array name to appear in more than one NAMELIST declaration. Check for possible conflicts with subroutine or function names. Severity Level: 8.

IEK069I THE COMMON STATEMENT HAS AN INVALID DELIMITER.

Check to see that the common statement is of the form:

```
(1)  COMMON /JIM/ list        (for labeled common)
(2)  COMMON // list           (for blank common)
     COMMON list
```

Severity Level: 8.

IEK070I THE EQUIVALENCE STATEMENT HAS A MISSING OR MISPLACED DELIMITER.

Check for unbalanced parentheses and misplaced commas. Severity Level: 8.

IEK071I THE EQUIVALENCE STATEMENT DOES NOT SPECIFY AT LEAST TWO VARIABLES TO BE EQUIVALENCED.

This diagnostic will be generated if commas and/or parentheses are misplaced. Severity Level: 8.

IEK072I THE EQUIVALENCE STATEMENT HAS AN INVALID VARIABLE NAME.

Dummy arguments are not legal in equivalence statements. Check all variable names for legality and absence of conflict in usage. If necessary, the MAP option can be used to give indication of conflict in usage. Severity Level: 8.

IEK073I THE EQUIVALENCE STATEMENT HAS A SUBSCRIPT WHICH IS NOT AN INTEGER CONSTANT.

Severity Level: 8.

IEK074I THE STATEMENT HAS A VARIABLE WITH MORE THAN SEVEN SUBSCRIPTS.

No array variable may have more than seven subscripts. Check commas and parentheses. Severity Level: 8.

IEK075I THE COMMON STATEMENT HAS A VARIABLE THAT HAS BEEN REFERENCED IN A PREVIOUS COMMON STATEMENT.

Variables and/or arrays may not appear in more than one COMMON declaration. Severity Level: 8.

IEK076I THE IMPLICIT STATEMENT IS NOT THE FIRST STATEMENT IN A MAIN PROGRAM OR THE SECOND STATEMENT IN A SUBPROGRAM.

Severity Level: 8.

IEK077I THE IMPLICIT STATEMENT HAS A MISPLACED DELIMITER IN THE TYPE SPECIFICATION FIELD.

Check to see that the parentheses are balanced and that commas appear in the proper places. Severity Level: 8.

IEK078I THE IMPLICIT STATEMENT HAS AN INVALID TYPE.

The type DOUBLE PRECISION cannot appear in the IMPLICIT statement. Make sure that the type declared has a valid standard or optional length specification. The types and lengths are as follows:

Type	Length	Standard Length
INTEGER	2 or 4	4
REAL	4 or 8	4
COMPLEX	8 or 16	8
LOGICAL	1 or 4	4

IEK079I THE IMPLICIT STATEMENT HAS A MISSING LETTER SPECIFICATION.

Insert the omitted specification. Severity Level: 8.

IEK080I THE IMPLICIT STATEMENT HAS AN INVALID LETTER SPECIFICATION.

Only the characters A,B,C,...,Z,$ may appear in an IMPLICIT declaration. Severity Level: 8.

IEK081I THE IMPLICIT STATEMENT HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK082I THE IMPLICIT STATEMENT DOES NOT END WITH A RIGHT PARENTHESIS.

Severity Level: 8.

IEK083I THE IMPLICIT STATEMENT HAS A MISPLACED DELIMITER IN ITS PARAMETER FIELD.

Check the format of the statement in question, noting the placement of parentheses and commas. Severity Level: 8.

IEK084I THE IMPLICIT STATEMENT CONTAINS A LITERAL FIELD.

There should not be any primes or Hollerith (wH) specifications in the IMPLICIT statements. Severity Level: 8.

IEK086I THE COMMON STATEMENT SPECIFIES A NONVARIABLE TO BE ENTERED.

Only variable or array names may be specified in a common statement. Check for conflicts in name usage. Severity Level: 8.

IEK087I THE COMMON STATEMENT SPECIFIES A NONVARIABLE COMMON BLOCK NAME.

Common block names must follow the normal FORTRAN rules for variable names. Severity Level: 8.

IEK088I A DUMMY ARGUMENT IN A SUBPROGRAM MAY NOT BE IN COMMON.

Severity Level: 8.

IEK090I THE EXTERNAL STATEMENT HAS A NONVARIABLE DECLARED AS EXTERNAL.

Names of external functions or subroutines must correspond to the normal FORTRAN rules for naming variables. Severity Level: 8.

IEK091I THE EXTERNAL STATEMENT HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK092I THE TYPE STATEMENT MULTIPLY DEFINES THE VARIABLE.

Severity Level: 8.

IEK093I THE TYPE STATEMENT HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK094I THE TYPE STATEMENT HAS A NONVARIABLE TO BE TYPED.

Look for a variable name which does not conform to the FORTRAN standards and check for the delimiters being properly placed. Severity Level: 8.

IEK095I THE TYPE STATEMENT HAS THE WRONG LENGTH FOR THE GIVEN TYPE.

See IEK078I for types and associated legal lengths. Severity Level: 8.

IEK096I THE TYPE STATEMENT HAS A MISSING DELIMITER.

Severity Level: 8.

IEK101I THE DO STATEMENT OR IMPLIED DO HAS AN INVALID DELIMITER.

Check the format of the DO, implied or stated, and note where the commas and equal sign are located. Severity Level: 8.

IEK102I THE BACKSPACE/REWIND/ENDFILE STATEMENT HAS AN INVALID DELIMITER.

There should be no delimiters except blanks in any of these statements. Severity Level: 8.

IEK104I THE BACKSPACE/REWIND/ENDFILE STATEMENT HAS A DATA SET REFERENCE NUMBER THAT IS EITHER A NONINTEGER OR AN ARRAY NAME.

The data set reference number must be an unsigned integer constant or an unsubscripted integer variable that is of length 4. Severity Level: 8.

IEK109I THE PAUSE STATEMENT HAS A MISPLACED DELIMITER.

The pause statement should contain (1) no delimiters, or (2) a literal constant enclosed in single quotes (e.g., 'HERE'), or (3) an integer constant. Severity Level: 8.

IEK110I THE PAUSE STATEMENT SPECIFIES A VALUE WHICH IS NEITHER A LITERAL NOR AN INTEGER CONSTANT.

Severity Level: 8.

IEK111I THE PAUSE STATEMENT HAS MORE THAN 255 CHARACTERS IN ITS LITERAL FIELD.

The most probable cause of this error is a missing second prime. Severity Level: 8.

IEK112I THE DICTIONARY HAS OVERFLOWED.

The program is too complex and should be divided into smaller sections.

Severity level: 16.

IEK115I THE VARIABLE RETURN STATEMENT HAS NEITHER AN INTEGER CONSTANT NOR VARIABLE FOLLOWING THE KEYWORD.

This error is most likely caused by an invalid variable name following the word RETURN. Severity Level: 8.

IEK116I THE DO STATEMENT OR IMPLIED DO HAS AN INVALID PARAMETER.

The DO variable must be an unsubscripted integer variable. In addition, the initial value, test value, and increment value must be represented by unsigned integer constants, or unsigned, nonsubscripted integer variables. The values of these parameters must be nonnegative and greater than zero. Severity Level: 8.

IEK117I THE BLOCK DATA STATEMENT HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK120I THE BLOCK DATA STATEMENT WAS NOT THE FIRST STATEMENT OF THE SUBPROGRAM.

Severity Level: 8.

IEK121I THE DATA STATEMENT HAS A VARIABLE WHICH HAS A NONALPHABETIC FIRST CHARACTER.

This problem can occur from the misplacement of commas and/or slashes. Check the format of the data statement. Severity Level: 8.

IEK122I THE DATA STATEMENT CONTAINS A SUBSCRIPTED VARIABLE WHICH HAS NOT BEEN DEFINED AS AN ARRAY.

This may result from a missing dimension or type statement which defines the variable as an array. Either the subscript should be deleted from the variable in question, or the variable should be defined as an array. The data statement must follow the definition statements. Severity Level: 8.

IEK123I THE DATA STATEMENT HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK124I THE DATA STATEMENT HAS A VARIABLE WITH AN INVALID INTEGER SUBSCRIPT.

The subscript must be only integer constants separated by commas. Severity Level: 8.

IEK125I THE DATA STATEMENT HAS A VARIABLE WITH A SUBSCRIPT THAT CONTAINS AN INVALID DELIMITER.

Severity Level: 8.

IEK129I THE STATEMENT CONTAINS AN INVALID DATA CONSTANT.

The constant must be valid for its designated class and type. Note that no conversion is done in DATA statements. Severity Level: 8.

IEK132I THE DATA STATEMENT HAS AN INVALID DELIMITER IN ITS INITIALIZATION VALUES.

Severity Level: 8.

IEK133I THE DO STATEMENT CANNOT FOLLOW A LOGICAL IF STATEMENT.

To solve this problem, change the DO to a "GO TO n", where "n" is the statement label of the DO located elsewhere in the program. Severity Level: 8.

IEK134I THE DO STATEMENT HAS AN INVALID INTEGER DO VARIABLE.

The DO variable must be a nonsubscripted variable. Severity Level: 8.

IEK135I THE DO STATEMENT OR IMPLIED DO HAS AN INVALID TEST VALUE.

The test value must be greater than zero and less than (2**31)-1. It must be in the form of an unsigned integer constant or an unsigned nonsubscripted integer variable. Severity Level: 8.

IEK136I THE NUMBER OF NESTED DO'S EXCEEDS THE COMPILER LIMIT.

The maximum number of levels for nesting is 25. Severity Level: 8.

IEK137I THE DO STATEMENT OR IMPLIED DO HAS AN INVALID INCREMENT VALUE.

Severity Level: 8.

IEK138I THE DO STATEMENT HAS A PREVIOUSLY DEFINED STATEMENT NUMBER SPECIFIED TO END THE DO RANGE.

Severity Level: 8.

IEK139I A LOGICAL IF IS FOLLOWED BY ANOTHER LOGICAL IF OR A SPECIFICATION STATEMENT.

Replace the second logical IF with a "GO TO n" statement, where "n" is the label of the second logical IF located elsewhere in the program or combine the LOGICAL IFs into one statement. The specification statement should be placed at the beginning of the program. Conditional specifications are not allowed. Severity Level: 8.

IEK140I THE IF STATEMENT BEGINS WITH AN INVALID CHARACTER.

Severity Level: 8.

IEK141I THE FORMAT STATEMENT DOES NOT END WITH A RIGHT PARENTHESIS.

Severity Level: 8.

IEK143I THE STATEMENT FUNCTION HAS AN ARGUMENT WHICH IS NOT A VARIABLE.

Arguments in statement functions must be nonsubscripted variables. The naming of variables must correspond to the standard FORTRAN conventions. Severity Level: 8.

IEK144I THE STATEMENT FUNCTION HAS MORE THAN 20 ARGUMENTS.

Severity Level: 8.

IEK145I THE STATEMENT FUNCTION HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK146I THE STATEMENT FUNCTION HAS A MISPLACED EQUAL SIGN.

The format of the statement function is: name (arg, ..., arg) = exp The expression cannot contain any equal signs, and there must be at least one dummy argument. Severity Level: 8.

IEK147I A STATEMENT FUNCTION DEFINITION MUST PRECEDE THE FIRST EXECUTABLE STATEMENT.

Severity Level: 8.

IEK148I THE DIMENSIONED ITEM HAS A NONINTEGER SUBSCRIPT.

The construction of any subscript must follow the rules outlined in the FORTRAN specifications in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515. Severity Level: 8.

IEK149I A VARIABLE TO BE DIMENSIONED USING ADJUSTABLE DIMENSIONS MUST HAVE BEEN PASSED AS AN ARGUMENT AND MUST NOT APPEAR IN COMMON.

Severity Level: 8.

IEK150I THE DIMENSIONED ITEM HAS AN INVALID DELIMITER.

Severity Level: 8.

IEK151I THE STATEMENT SPECIFIES A NONVARIABLE TO BE DIMENSIONED.

The variable name does not conform to the IBM FORTRAN standard. Severity Level: 8.

IEK152I THE SUBPROGRAM STATEMENT HAS AN INVALID DELIMITER IN THE ARGUMENT LIST.

The arguments must be separated by commas, and no other delimiter other than a blank is allowed to precede or follow the comma. Severity Level: 8.

IEK153I THE STATEMENT HAS AN INVALID NAME SPECIFIED AS A FUNCTION REFERENCE.

This diagnostic will result if the function has been improperly defined or if the type of name used as a reference does not agree with the type of name used in the definition of the function. Both the implicit and/or actual declarations of type must match. Severity Level: 8.

IEK156I THE I/O STATEMENT HAS AN INVALID NAME PRECEDING THE EQUAL SIGN.

Severity Level: 8.

IEK157I THE I/O STATEMENT HAS A NONVARIABLE SPECIFIED AS A LIST ITEM.

No function references or arithmetic expressions may appear in the I/O list. Severity Level: 8.

IEK158I THE I/O STATEMENT HAS AN IMPROPER PAIRING OF PARENTHESES IN AN IMPLIED DO, OR A NONINTEGRAL INDEX.

Severity Level: 8.

IEK159I THE FORMAT STATEMENT DOES NOT HAVE A STATEMENT NUMBER.

Severity Level: 4.

IEK160I THE I/O STATEMENT HAS AN INVALID DELIMITER IN THE PARAMETERS.

Severity Level: 8.

IEK161I THE I/O STATEMENT HAS A DUPLICATE PARAMETER.

Severity Level: 8.

IEK163I THE I/O STATEMENT HAS AN ARRAY WHICH IS NOT DIMENSIONED.

Severity Level: 8.

IEK164I THE I/O STATEMENT HAS AN ARITHMETIC EXPRESSION OR A FUNCTION NAME SPECIFIED AS A LIST ITEM.

Severity Level: 8.

IEK165I THE I/O STATEMENT HAS A PARAMETER WHICH IS NOT AN ARRAY AND NOT A NAMELIST NAME.

This diagnostic can result if there is an undefined function reference or reference to an undimensioned array. Severity Level: 8.

IEK166I THE I/O STATEMENT HAS A NONINTEGER CONSTANT OR VARIABLE REPRESENTING THE DATA SET REFERENCE NUMBER.

The data set reference number must be an unsigned integer constant or an integer variable of length 4. Severity Level: 8.

IEK167I THE STATEMENT HAS AN INVALID USE OF A STATEMENT FUNCTION NAME.

This error can result if the statement function referenced has not yet been defined. There may be other causes. Severity Level: 8.

IEK168I THE STATEMENT SPECIFIES AS A SUBPROGRAM NAME A VARIABLE WHICH HAS BEEN PREVIOUSLY USED AS A NONSUBPROGRAM NAME.

If the variable name duplicates the subprogram name, either change the variable name or the subprogram name and all references to the one changed. Use of the XREF option can be helpful in locating references if the program contains numerous statements. Severity Level: 8.

IEK169I THE DIRECT ACCESS I/O STATEMENT MAY NOT SPECIFY A NAMELIST NAME.

Severity Level: 8.

IEK170I THE DIRECT ACCESS I/O STATEMENT HAS A NONINTEGER SPECIFYING THE RECORD'S RELATIVE POSITION.

Severity Level: 8.

IEK171I THE NAME SPECIFIED FOR AN ENTRY POINT HAS ALREADY BEEN USED AS EITHER A VARIABLE SUBROUTINE OR FUNCTION NAME.

Severity Level: 8.

IEK176I THE I/O STATEMENT CONTAINS INVALID SYNTAX IN ITS IMPLIED DO.

This diagnostic will result when there is invalid syntax, and also if there are more than 20 implied DOs in the I/O statement. Severity Level: 8.

IEK192I THE STATEMENT HAS A LABEL WHICH IS SPECIFIED AS BOTH THE LABEL OF A FORMAT STATEMENT AND THE OBJECT OF A BRANCH.

Severity Level: 8.

IEK193I THE STATEMENT NUMBER HAS BEEN PREVIOUSLY DEFINED.

Severity Level: 8.

IEK194I THE TYPE STATEMENT HAS A MISSING DELIMITER IN THE INITIALIZATION VALUES.

Severity Level: 8.

IEK197I THE STOP STATEMENT HAS A NONINTEGER CONSTANT AFTER THE KEYWORD.

The constant must be a string of 1-5 decimal digits. Severity Level: 8.

IEK199I THE SUBROUTINE OR FUNCTION STATEMENT WAS NOT THE FIRST STATEMENT.

No statements other than comment statements may precede the SUBROUTINE or FUNCTION statement. Severity Level: 8.

IEK200I QUOTE LITERALS MAY APPEAR ONLY IN CALL, DATA INITIALIZATION, FUNCTION, AND FORMAT STATEMENTS.

Severity Level: 8.

IEK202I THE STATEMENT HAS A VARIABLE WHICH HAS BEEN PREVIOUSLY DIMENSIONED. THE INITIAL DIMENSION FACTORS ARE USED.

Severity Level: 4.

IEK203I AN ENTRY STATEMENT MUST NOT APPEAR IN A MAIN PROGRAM. THE STATEMENT IS IGNORED.

Severity Level: 8.

IEK204I THE STOP STATEMENT HAS AN INVALID DELIMITER.

The only valid delimiters are blanks. Severity Level: 4.

IEK205I THE ASSIGNED OR COMPUTED GO TO HAS AN INVALID ELEMENT FOLLOWING THE CLOSING PARENTHESIS.

Severity Level: 4.

IEK206I THE STATEMENT HAS A NONSUBSCRIPTED ARRAY ITEM.

Severity Level: 4.

IEK207I THE CONTINUE STATEMENT DOES NOT END AFTER THE KEYWORD CONTINUE.

This error message will result if the statement following the CONTINUE has indications that it is a continuation, or if there is garbage following the word CONTINUE. Severity Level: 4.

IEK208I THE CONTINUE STATEMENT DOES NOT HAVE A STATEMENT NUMBER.

Severity Level: 4.

IEK209I THE STATEMENT HAS AN OCTAL CONSTANT SPECIFIED AS AN INITIAL VALUE. THE VALUE IS REPLACED BY ZERO.

This diagnostic will result if the letter "O" is inadvertently punched in place of a leading zero in a constant. If an octal constant is required, convert it to the appropriate hexadecimal constant. Severity Level: 4.

IEK211I THE STATEMENT HAS A COMPLEX CONSTANT WHOSE REAL CONSTANTS DIFFER IN LENGTH.

Both parts of the constant must be either REAL*4 or REAL*8, but not a combination of the two. Severity Level: 4.

IEK212I THE BLOCK DATA SUBPROGRAM CONTAINS EXECUTABLE STATEMENT-(S). THE EXECUTABLE STATEMENT(S) IS IGNORED.

Severity Level: 4.

IEK222I THE EXPRESSION HAS A LITERAL WITH A MISSING DELIMITER.

This diagnostic can result from either a missing delimiter or use of an invalid delimiter. Severity Level: 4.

IEK224I THE STATEMENT AFTER AN ARITHMETIC IF, GO TO, OR RETURN HAS NO LABEL.

Because there is no label on this statement, there is no path to it. It cannot be executed. Supply a label or make sure there is a path. Severity Level: 4.

IEK225I A LABEL APPEARS ON A NONEXECUTABLE STATEMENT. THE LABEL IS IGNORED.

Severity Level: 4.

IEK226I THE STATEMENT HAS A VARIABLE WITH MORE THAN SIX CHARACTERS. THE RIGHTMOST CHARACTERS ARE TRUNCATED.

This can result from a missing delimiter. Either truncate the name or insert the delimiter. Severity Level: 4.

IEK229I ALL THE ARGUMENTS OF AN ARITHMETIC STATEMENT FUNCTION ARE NOT USED IN THE DEFINITION.

The expression to the right of the equal sign should contain as many distinct variables as there are dummy arguments. Severity Level: 4.

IEK302I THE EQUIVALENCE STATEMENT HAS EXTENDED COMMON BACKWARDS.

Look at all implicit and explicit equivalencing or assignment statements to determine where the error occurred. Severity Level: 8.

IEK303I THE EQUIVALENCE STATEMENT CONTAINS AN ARRAY WHICH IS NOT DIMENSIONED.

Mention of an array in an equivalence statement must include a subscript. Severity Level: 8.

IEK304I THE EQUIVALENCE STATEMENT HAS LINKED BLOCKS OF COMMON TOGETHER.

Severity Level: 8.

IEK305I THE EQUIVALENCE STATEMENT CONTAINS AN ARRAY WITH A SUBSCRIPT WHICH IS OUT OF RANGE.

Severity Level: 4.

IEK306I THE EQUIVALENCE STATEMENT HAS AN INCONSISTENCY.

This diagnostic will result if the equivalence statement contradicts itself or any previously established equivalencies, implicit or explicit. Severity Level: 8.

IEK307I THE DATA STATEMENT CONTAINS A VARIABLE THAT IS NOT REFERENCED.

Severity Level: 4.

IEK308I THE EQUIVALENCE STATEMENT HAS EQUIVALENCED TWO VARIABLES IN THE SAME COMMON BLOCK.

Severity Level: 8.

IEK312I THE EQUIVALENCE STATEMENT CONTAINS AN EXTERNAL REFERENCE.

The externally referenced name should be either deleted or corrected. Severity Level: 8.

IEK314I THE EQUIVALENCE STATEMENT MAY CAUSE WORD-BOUNDARY ERRORS.

IEK315I THE EQUIVALENCE STATEMENT WILL CAUSE WORD-BOUNDARY ERRORS.

Variables should be arranged in fixed descending order according to length. If this cannot be done, then proper alignment should be forced using dummy variables. The MAP option can be used for information on the variables and their relative addresses. Severity Level: 4.

IEK317I THE BLOCK DATA PROGRAM DOES NOT CONTAIN A COMMON STATEMENT.

Severity Level: 8.

IEK318I THE DATA STATEMENT IS USED TO ENTER DATA INTO COMMON OUTSIDE A BLOCK DATA SUBPROGRAM.

The variable referenced will have to be DATA-initialized in a BLOCK DATA program or in an assignment statement. Severity Level: 8.

IEK319I DATA IS ENTERED INTO A LOCAL VARIABLE IN A BLOCK DATA PROGRAM.

Check the spelling of all initialized variables. Delete the local variable or make sure that it is included in a COMMON block. Severity Level: 8.

IEK320I DATA MAY NOT BE ENTERED INTO A VARIABLE WHICH HAS BEEN PASSED AS AN ARGUMENT.

Dummy arguments cannot be data initialized. Their value may be changed using an assignment statement, however. Data initialize the variable in the calling program or use an assignment statement. Severity Level: 8.

IEK322I THE COMMON STATEMENT MAY CAUSE WORD-BOUNDARY ERRORS.

IEK323I THE COMMON STATEMENT WILL CAUSE A WORD-BOUNDARY ERROR.

Ideally, the variables in the COMMON block should be arranged in fixed descending order according to length. If this is not possible, the proper alignment should be forced using dummy variables. Severity Level: 4.

IEK332I THE STATEMENT NUMBER IS UNDEFINED.

Severity Level: 8.

IEK334I THE COMMON STATEMENT HAS A VARIABLE WITH A VARIABLE DIMENSION.

The subscript used in COMMON statements must be 1-7 unsigned integer constants separated by commas. Severity Level: 8.

IEK350I THE DATA STATEMENT HAS A MISSING PARENTHESIS.

This can result from a misplacement of quotes. Severity Level: 8.

IEK351I THE DATA INITIALIZATION VALUE IS LARGER THAN THE VARIABLE OR ARRAY ELEMENT. TRUNCATION OR SPILL WILL OCCUR.

An array or variable was initialized with a constant whose length was greater than that of the variable or array element. If the constant was specified as the first element in an array which was not subscripted in the data statement, then part of the constant will spill over into the other array element(s). All other cases result in truncation to the appropriate length. Severity Level: 4.

IEK352I THE DATA STATEMENT HAS TOO MANY INITIALIZATION VALUES.

There must be a one-to-one correspondence between the initialization values and the total number of items to be initialized. Severity Level: 4.

IEK353I THE DIMENSION STATEMENT HAS A VARIABLE WHICH HAS A SUBSCRIPT OF REAL MODE.

Severity Level: 8.

IEK354I A VARIABLE TO BE DIMENSIONED USING ADJUSTABLE DIMENSIONS MUST HAVE BEEN PASSED AS AN ARGUMENT AND MUST NOT APPEAR IN COMMON.

Severity Level: 8.

IEK355I ADCON TABLE EXCEEDED.

This will result if an expression is too long and/or complex to be evaluated. Restructure the statement into a series of less complex statements. Severity Level: 16. (Note: For any error of severity level 16, the output should be taken to a Computing Center consultant for advice.)

IEK356I A PARAMETER CANNOT ALSO BE IN COMMON.

Dummy arguments cannot be in common. Severity Level: 8.

IEK357I THE ARRAY HAS AN INCORRECT ADJUSTABLE DIMENSION.

Severity Level: 8.

IEK358I THE ADJUSTABLE DIMENSION IS NOT PASSED AS AN ARGUMENT OR IN COMMON.

Severity Level: 8.

IEK500I THE ARGUMENT TO A FORTRAN-SUPPLIED FUNCTION IS OF THE WRONG TYPE. THE FUNCTION IS ASSUMED TO BE USER-DEFINED.

Check the argument type required by the standard FORTRAN function. If the function is to be user-supplied, make sure that it appears in an EXTERNAL statement. Severity Level: 4.

IEK501I THE EXPRESSION HAS A COMPLEX EXPONENT.

Severity Level: 8.

IEK502I THE EXPRESSION HAS A BASE WHICH IS COMPLEX, BUT THE EXPONENT IS NONINTEGER.

Severity Level: 8.

IEK503I A NONSUBSCRIPTED ARRAY ITEM APPEARS IMPROPERLY WITHIN A FUNCTION REFERENCE OR A CALL.

Severity Level: 8.

IEK504I THE BASE AND/OR EXPONENT IS A LOGICAL VARIABLE.

The base must be real, complex, or integer. The exponent may be real or integer. Severity Level: 8.

IEK505I THE INPUT/OUTPUT STATEMENT REFERS TO THE STATEMENT NUMBER OF A NONFORMAT STATEMENT.

Severity Level: 8.

IEK506I THERE IS A MISSING OPERAND PRECEDING A RIGHT PARENTHESIS.

Severity Level: 8.

IEK507I A NONSUBSCRIPTED ARRAY ITEM IS USED AS AN ARGUMENT TO AN IN-LINE FUNCTION.

Severity Level: 8.

IEK508I THE NUMBER OF ARGUMENTS TO AN IN-LINE FUNCTION IS INCORRECT.

Severity Level: 8.

IEK509I THE PROGRAM DOES NOT END WITH A STOP, RETURN, OR GO TO STATEMENT.

It is not possible to merely fall through to the END statement. Severity Level: 4.

IEK510I THE EXPRESSION HAS A LOGICAL OPERATOR WITH A NONLOGICAL OPERAND.

Severity Level: 8.

IEK512I THE LOGICAL IF DOES NOT CONTAIN A LOGICAL EXPRESSION.

Severity Level: 8.

IEK515I THE EXPRESSION HAS A RELATIONAL OPERATOR WITH A COMPLEX OPERAND.

If it is necessary to use the operand with the relational operator, then equivalence a real array of two elements to the operand and then proceed. Severity Level: 8.

IEK516I THE ARITHMETIC IF CONTAINS A COMPLEX EXPRESSION.

See IEK515I. Severity Level: 8.

IEK520I THERE IS A COMMA IN AN INVALID POSITION.

Severity Level: 8.

IEK521I THE EXPRESSION HAS AT LEAST ONE EXTRA RIGHT PARENTHESIS.

Severity Level: 8.

IEK522I THE EXPRESSION HAS AT LEAST ONE TOO FEW RIGHT PARENTHESES.

Severity Level: 8.

IEK523I THE EQUAL SIGN IS IMPROPERLY USED.

Severity Level: 8.

IEK524I THE EXPRESSION HAS AN OPERATOR MISSING AFTER A RIGHT PARENTHESIS.

Severity Level: 8.

IEK525I THE EXPRESSION USES A LOGICAL OR RELATIONAL OPERATOR INCORRECTLY.

This diagnostic can result from use of an invalid operand expression. Operators must be preceded and followed by periods. Severity Level: 8.

IEK529I A FUNCTION NAME APPEARING AS AN ARGUMENT HAS NOT BEEN DECLARED EXTERNAL.

Severity Level: 8.

IEK530I THE EXPRESSION HAS A VARIABLE WITH AN IMPROPER NUMBER OF SUBSCRIPTS.

This can result from a misplaced delimiter in the subscript field. There should be as many subscripts as there are in the associated dimensioning statement. Severity Level: 8.

IEK531I THE EXPRESSION HAS A STATEMENT FUNCTION REFERENCE WITH AN IMPROPER NUMBER OF ARGUMENTS.

Severity Level: 8.

IEK541I AN ARGUMENT TO A LIBRARY FUNCTION HAS AN INVALID TYPE.

Consult the list of library functions in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515, to determine the required type of the arguments to each function. Severity Level: 8.

IEK542I A LOGICAL EXPRESSION APPEARS IN AN INVALID CONTEXT.

Severity Level: 8.

IEK550I PUSHDOWN, ADCON, OR ASF ARGUMENT TABLE EXCEEDED.

The program is too complex and should be divided into smaller sections.

Severity Level: 16.

IEK552I SOURCE PROGRAM IS TOO LARGE.

Subdivide the program. Severity Level: 16.

IEK570I TABLE EXCEEDED. OPTIMIZATION DOWNGRADED.

Program is too large to permit optimization. OPT=1 register allocation only is performed. Severity Level: 0.

IEK580I COMPILER ERROR.

Severity Level: 16.

IEK600I INTERNAL COMPILER ERROR. LOGICALLY IMPOSSIBLE BRANCH TAKEN IN A COMPILER SUBROUTINE.

Severity Level: 16.

IEK610I THE STATEMENT NUMBER OR GENERATED LABEL IS UNREACHABLE.

This message will be generated only if OPT=2 is specified. The message can be caused by an unlabeled STOP, RETURN, or GO TO which immediately follows another STOP, RETURN, or GO TO. It may also be generated if an unlabeled statement follows an arithmetic IF. Severity Level: 4.

IEK620I THE STATEMENT LABEL OR GENERATED LABEL IS A MEMBER OF AN UNREACHABLE LOOP.

This message will only be generated if OPT=2 was specified. Control statements should indicate correct target branches. Severity Level: 4.

IEK630I INTERNAL TOPOLOGICAL ANALYSIS TABLE EXCEEDED.

The program is too complex and should be divided into smaller sections.

Severity Level: 16.

IEK640I COVERAGE BY BASE REGISTER 12 IN OBJECT MODULE EXCEEDED.

The program is too large and should be divided into smaller sections.

Severity Level: 16.

IEK650I INTERNAL ADCON TABLE EXCEEDED.

The program is too large and should be divided into smaller sections.

Severity Level: 16.

IEK660I INTERNAL COMPILER ERROR. TEMPORARY FETCHED BUT NEVER STORED.

Severity Level: 16.

IEK661I INTERNAL COMPILER ERROR. UNABLE TO FREE A REGISTER.

Severity Level: 16.

IEK662I INTERNAL COMPILER ERROR. TEMPORARY NOT ENTERED IN ASSIGNMENT TABLE.

Severity Level: 16.

IEK670I LOGICALLY IMPOSSIBLE BRANCH TAKEN IN A COMPILER SUBROUTINE.

Severity Level: 16.

IEK671I LOGICALLY IMPOSSIBLE BRANCH TAKEN IN A COMPILER SUBROUTINE.

Severity Level: 16.

IEK710I THE FORMAT STATEMENT SPECIFIES A FIELD WIDTH OF ZERO.

Severity Level: 8.

IEK720I THE FORMAT STATEMENT CONTAINS AN INVALID CHARACTER.

Severity Level: 8.

IEK730I THE FORMAT STATEMENT HAS UNBALANCED PARENTHESES.

Severity Level: 8.

IEK740I THE FORMAT STATEMENT HAS NO BEGINNING LEFT PARENTHESIS.

Severity Level: 8.

IEK750I THE FORMAT STATEMENT SPECIFIES A COUNT OF ZERO FOR A LITERAL FIELD.

Severity Level: 8.

IEK760I THE FORMAT STATEMENT CONTAINS A MEANINGLESS NUMBER.

Severity level: 8.

IEK770I THE FORMAT STATEMENT HAS A MISSING DELIMITER.

Severity Level: 8.

IEK780I THE FORMAT STATEMENT CONTAINS A NUMERIC SPECIFICATION GREATER THAN 255.

No replication or specification number can be greater than 255 in a FORMAT statement. Severity Level: 8.

IEK790I THE FORMAT STATEMENT CONTAINS GROUP FORMAT SPECIFICATIONS NESTED TO A LEVEL GREATER THAN TWO.

This diagnostic is generated when the compiler detects more than two left parentheses without an intervening right parenthesis in a FORMAT statement. Severity Level: 8.

IEK800I SOURCE PROGRAM IS TOO LARGE.

Severity Level: 16.

IEK901I NO SOURCE INPUT FOUND AT END OF FILE ON SCARDS.

No source input is detected by the compiler. Severity Level: 4.

IEK1000I INTERNAL COMPILER ERROR.

Severity Level: 4.

October 1983

VS FORTRAN

The public file *FORTRANVS contains Version 2, Release 2.0 of the IBM VS FORTRAN compiler. This version runs in native MTS mode and is located in shared virtual memory.

VS FORTRAN meets the most recent standard published by the American National Standards Institute (ANSI), FORTRAN 77. It also supports IBM extensions to the language and contains features and extensions that are not available with FORTRAN 66 compilers.

This compiler can produce object programs that utilize the IBM 3090-400 Vector Facility. This will significantly speed up large-scale numerical processing, especially arrays inside of DO-loops.

The MTS version in *FORTRANVS supports all documented features of the IBM version except asynchronous I/O and keyed I/O.

VS FORTRAN is documented in the following IBM publications:

VS FORTRAN Version 2: Language and Library Reference, Release 1.1 or 2.0, form SC26-4221.
VS FORTRAN Version 2: Programming Guide, Release 1.1 or 2.0, form SC26-4222.

In addition, the FORTRAN 77 standard is documented in Programming Language FORTRAN, ANSI Standard X3.9-1978.

COMPILING A VS FORTRAN PROGRAM

The VS FORTRAN compiler is run by specifying the following command:

```
$RUN *FORTRANVS SCARDS=source SPUNCH=object SPRINT=list SERCOM=errs
     0=include 1=deck PAR=options
```

where

"source" is the file containing the VS FORTRAN source program;
"object" is the file in which the object code will be stored;
"list" is the file in which the compiler will write source listings, error messages, and optionally, object listings, storage maps, and cross-reference listings;
"errs" is the file to which error messages will be written;
"include" is a file containing additional source statements to be included in the source program (see the section "The

INCLUDE Feature" below);
"deck" is the file in which the deck code will be stored;
"options" is one or more compiler options (see the section "Compiler Options" below).

The default assignments are as follows:

SCARDS - defaults to *SOURCE* for program input. If SCARDS is not assigned and *SOURCE* is assigned to the terminal, the compiler will read the source program from the terminal.
SPRINT - defaults to *SINK* for source listings. If SPRINT is not assigned and *SINK* is assigned to the terminal, no compiler output listings will be produced.
SPUNCH - defaults to -LOAD for the object file. If SPUNCH is not assigned, -LOAD will be emptied before the object program is written; if SPUNCH is explicitly assigned to a file, that file is not emptied before the object program is written.
SERCOM - defaults to *MSINK* for messages.
0 - is optional and has no default.
1 - is optional and is identical to SPUNCH.

EXECUTING A VS FORTRAN PROGRAM

To execute the compiled program, the following $RUN command should be given:

```
$RUN object
```

The usual FORTRAN library default I/O units apply, that is:

```
5 = *SOURCE*
6 = *SINK*
```

Note: If the program is compiled with either the VECTOR or IL(NODIM) options or if the program is to use special mathematical functions, the IBM SDUMP subroutine, or the IBM extended error-handing subroutines, the *FORTRANLIB library must be concatenated to the object program, i.e.,

```
$RUN object+*FORTRANLIB
```

See "VS Fortran I/O Library" below for details.

COMPILER OPTIONS

Compile-time options may be included in the PAR field of the $RUN command, or they may be included in the source file in an @PROCESS

statement, as described at the end of this section. More than one option may be included, but each option must be separated from the next by a comma or blank. The available options are described below. See the IBM publication, VS FORTRAN Version 2: Programming Guide, for more information. In the following list, the default is given for each option and accepted abbreviations are shown.

AUTODBL(value) Default: AUTODBL(NONE)

The AUTODBL option specifies the automatic conversion of single-precision calculations to double precision, double-precision calculations to extended precision. For more information concerning "value", consult "Using the Automatic Precision Increase Facility -- AUTODBL Option" in the IBM publication, VS FORTRAN Version 2: Programming Guide. AUTODBL may be abbreviated as AD.

CHARLEN(number) Default: 32767

The CHARLEN option specifies the maximum length for any CHARACTER variable, CHARACTER array element, or CHARACTER function. The value of "number" must be an integer in the range of 1 to 32767. If an illegal value is specified, the default of 32767 will be used. CHARLEN may be abbreviated as CL.

CI(number1,number2,...) Default: None

The CI option specifies the identification numbers of the conditional INCLUDE statements to be processed. The values supplied must be in the range of 1 to 255. For further details, see the section "The INCLUDE Feature" below and the description of the INCLUDE statement in the IBM publication, VS FORTRAN Version 2: Language and Library Reference.

DC(name1,name2,...) Default: None

The DC option specifies the names of COMMON blocks that are to be allocated at execution time. This option allows the specification of very large (i.e., more than one megabyte) COMMON blocks.

DECK / NODECK Default: NODECK

The DECK option specifies that the object program is to be written on logical I/O unit 1. This option is unnecessary because, by default, an identical object program is written on SPUNCH.

DIRECTIVE(trigger-constant) / NODIRECTIVE[(trigger-constant)]
Default: NODIRECTIVE

The DIRECTIVE option specifies whether or not the processing of selected comments as vector directive statements is enabled or disabled. The DIRECTIVE option can only be specified in an @PROCESS statement once for each compilation unit. Refer to "Vector Directives" in the IBM publication, VS FORTRAN Version 2:

Programming Guide, for more information on vector directive statements.

"trigger-constant" is a character constant used to identify directives in comment statements. If the constant is specified with NODIRECTIVE option, the processing of a particular vector directive statement will be disabled.

FIPS(S|F) / NOFIPS Default: NOFIPS

The FIPS option controls the flagging of statements in the source program that are not defined in the FORTRAN 77 standard (see Programming Language FORTRAN, ANSI Standard X3.9-1978).

FIPS(S) requests the compiler to flag all those language elements not included in the ANSI Subset standard.

FIPS(F) requests the compiler to flag all language elements not included in the ANSI Full standard. See the IBM pubication, VS FORTRAN Version 2: Language and Library Reference, Appendix A, for a list of items flagged when FIPS(F) is used.

FIPS flagging is ignored if the FREE or LANGLVL(66) options are in effect.

FLAG(I|W|E|S) Default: FLAG(I)

The FLAG option specifies the level of diagnostic messages to be written. There are four levels of messages: information messages, warning messages (those generating a return code of 4), error messages (those generating a return code of 8), and severe error messages (those generating a return code of 12 or higher).

FLAG(I) indicates that messages of all four levels are to be written.

FLAG(W) indicates that warning, error, and severe error messages are to be written.

FLAG(E) indicates that only error messages and severe error messages are to be written.

FLAG(S) indicates that only severe error messages are to be written.

FREE / FIXED Default: FIXED

The FREE option specifies that the source program is written in free format. The FIXED option specifies that the source program is written in fixed format (column-aligned). See the IBM publication, VS FORTRAN Version 2: Language and Library Reference, for further details.

GOSTMT / NOGOSTMT Default: NOGOSTMT

The GOSTMT option indicates that the internal statement numbers (ISNs) are to be generated in the object program for calling sequences to subroutines. The ISNs are used when generating a traceback map during program debugging. The NOGOSTMT option specifies that ISNs are not generated in the object program. GOSTMT may be abbreviated as GS and NOGOSTMT as NOGS.

ICA suboption / NOICA

The ICA option specifies whether intercompilation analysis is to be performed, specifies the files containing ICA information to be used or updated, and controls output from the intercompilation analyzer. ICA should be specified when there is a group of separately-compiled programs and subprograms to be executed together and there is a need to know if there are any conflicting external references. The available suboptions are:

USE(name1,name2,...)

USE specifies the names of the ICA files to be included in the analysis. This option can be repeated any number of times as long as the total number of files specified in the USE and UPDATE suboptions does not exceed nine (9).

"name" is the name of an ICA file containing entries describing interfaces between program units. The name can be a sequence of 1 to 8 alphanumeric characters and must begin with a letter. Names can be separated by commas or blanks. The actual file names are formed by appending the name with ICA.

UPDATE(name)

The ICA file name can be a sequence of 1 to 8 alphanumeric characters and must begin with a letter. The compiler automatically appends the file name with ".ICA".

MXREF / NOMXREF

This suboption specifies whether to produce external cross-reference listings. The default is MXREF.

CLEN / NOCLEN

This suboption specifies whether to check the length of named common blocks. The default is CLEN.

CVAR / NOCVAR

This suboption specifies whether usage information for variables in a named common block is to be collected. The default is NOCVAR.

MSG(NEW|NONE|ALL)

This suboption specifies the type of diagnostic messages to appear in the printout.

NEW specifies that only messages about the new compilations will be printed.

NONE specifies that only messages about deleting entries in an ICA file will be printed.

ALL specifies that all messages will be printed.

The default is NEW.

IL(DIM) / IL(NODIM) Default: IL(DIM)

The IL option specifies whether the code for adjustably-dimensioned arrays is to be placed inline - IL(DIM), or done via library call - IL(NODIM). Inline code may result in faster execution, but it does not check for user dimensioning errors. The library call method may result in slower execution, but it does check for such errors. IL(NODIM) may also be specified as NOIL.

If IL(NODIM) is specified, then the *FORTRANLIB library must be concatenated to the object program on the $RUN command.

LANGLVL(66|77) Default: LANGLVL(77)

The LANGLVL option specifies the language level of the source to be compiled, FORTRAN 77 or FORTRAN 66.

LINECOUNT(number) Default: LINECOUNT(60)

The LINECOUNT option specifies the number of lines to be printed per page of the source listing. The number must be in the range 5 to 32765. If an illegal value is specified, the default of 60 will be used. LINECOUNT may be abbreviated as LC.

LIST / NOLIST Default: NOLIST

The LIST option specifies that an object program listing is to be written on SPRINT. The listing consists of statements written in pseudo-assembly-language format. This option drastically increases the amount of printed output. Unless the user is familiar with machine code and can read hexadecimal dumps, the listing is generally not useful.

The LIST output is fully described under "Object Module Listing--LIST Output" in the IBM publication, VS FORTRAN Version 2: Programming Guide.

MAP / NOMAP Default: NOMAP

The MAP option specifies that a storage map is to be written on SPRINT. The map includes program variables, subroutines and functions called, statement labels, and common blocks. These symbols are listed with a coded description of the context in which they appear and their addresses in storage. Some of this information is also given by the XREF option.

NAME(name) Default: NAME(MAIN)

The NAME option specifies the name to be given to the main object program. This option is available only with LANGLVL(66). A PROGRAM source statement should be used to specify the program name for LANGLVL(77).

OBJECT / NOOBJECT Default: OBJECT

The OBJECT option specifies that the object program is to be written on SPUNCH. NOOBJECT specifies that no object program is to be produced. The object program produced by OBJECT is identical to that produced by DECK.

OPTIMIZE(0|1|2|3) / NOOPTIMIZE Default: OPTIMIZE(0)

The OPTIMIZE option indicates the optimization level to be used in generating the object code for the program. Four levels are available; the higher the level, the more efficient is the program, and the slower and more expensive the compilation. Through optimization techniques, the compiler can create a more efficient program both with respect to execution time and storage requirements.

OPTIMIZE(0) indicates that no optimization is to be performed; it is equivalent to NOOPTIMIZE. This level provides the fastest compile time, but the least efficient program. It is a good level for debugging a program or for checking program syntax.

OPTIMIZE(1) specifies a moderate level of local register and branch optimization without considering program loops.

OPTIMIZE(2) specifies full optimization of the entire program without moving code outside of loops if there is any possibility of this causing program errors.

OPTIMIZE(3) specifies full optimization. It is best suited for fully debugged programs ready for production use. This level is safe for most programs, except that for certain modules it may be necessary to reduce the level of optimization if problems should arise.

See the section "Program Efficiency" in the IBM publication, VS FORTRAN Version 2: Programming Guide, for further information about optimization.

Warning: The use of the Symbolic Debugging System (SDS) may not be completely reliable for debugging programs that have been optimized (level 1 or higher) as up-to-date copies of certain program variables may not be maintained in storage, but in registers, or certain variables such as loop counters may be eliminated all together.

RENT / NORENT Default: NORENT

The RENT option specifies that the compiler generate a reentrant object module for a program being compiled. Currently Fortran reentrant object modules are not supported in MTS.

SDUMP(ISN|SEQ) / NOSDUMP Default: SDUMP(ISN)

The SDUMP option specifies that symbolic dump information is to be generated.

SDUMP(ISN) specifies SDUMP tables be generated using internal statement numbers.

SDUMP(SEQ) specifies SDUMP tables be generated using sequence numbers in columns 73-80 of fixed-form source.

SDUMP may be abbreviated as SD, and NOSDUMP as NSD.

SOURCE / NOSOURCE Default: SOURCE

The SOURCE option specifies that a source listing is to be written on SPRINT. The NOSOURCE option specifies that source listings are not to be written at all.

SRCFLG / NOSRCFLG Default: SRCFLG

The SRCFLG option specifies that error messages appear in the source listing immediately after the statement causing an error. The NOSRCFLG option suppresses these error messages. SRCFLG may be abbreviated as SF and NOSRCFLG as NOSF.

Note: These error messages will always appear at the end of the source listing for each program unit and again in a summary display at the end of all the listings.

SXM / NOSXM Default: NOSXM

This option specifies that XREF or MAP listing output be formatted for a 72-character-wide terminal screen. The NOSXM option formats listing output for a printer. For more details, see "Using the SXM

Option" in IBM publication, VS FORTRAN Version 2: Programming Guide.

SYM / NOSYM Default: NOSYM

The SYM option generates SYM records in the object program. This allows the program to be debugged using the Symbolic Debugging System (SDS). See the section "Introduction to Debug Mode for FORTRAN" for further details.

TERMINAL / NOTERMINAL Default: TERMINAL

The TERMINAL option indicates that error messages are to be written on SPRINT at the end of the source listing for each program unit and again in a summary display at the end of all the listings. NOTERMINAL suppresses the summary display.

The TERMINAL option does not control error messages that are written to SERCOM.

TEST / NOTEST Default: NOTEST

The TEST option is the same as the SYM / NOSYM option in MTS.

TRMFLG / NOTRMFLG Default: TRMFLG

The TRMFLG option specifies that erroneous statements are to be displayed along with their associated error messages on SERCOM (if SERCOM is assigned to a terminal). The NOTRMFLG option suppresses the printing of the statements. TRMFLG may be abbreviated as TF and NOTRMFLG as NOTF.

Note: Error messages are always displayed on SERCOM.

VECTOR(options) / NOVECTOR Default: NOVECTOR

The VECTOR option invokes the vectorization process to produce programs to be run on the IBM System 3090 Vector Facility. For more information on VECTOR, consult Chapter 8, "Vectorizing Your Program," in the IBM publication, VS FORTRAN Version 2: Programming Guide.

The VECTOR option has several suboptions: LEVEL, REPORT, INTRINSIC, REDUCTION, and SIZE. OPTIMIZE(3) is required if LEVEL≥1 in the VECTOR option.

If VECTOR is specified, then the *FORTRANLIB library must be concatenated to the object program on the $RUN command.

XREF / NOXREF Default: NOXREF

The XREF option specifies that two cross-reference dictionaries be included in the output listings on SPRINT. The first is a listing

of program variables, subroutine names, and function names; the second is a listing of statement labels. Internal statement numbers for all references to symbols in the program are given in both dictionaries. Some of this information is also given by the MAP option.

Conflicting Compiler Options

The following table lists conflicting compiler options that create an error message if both are used. The table also reflects those options that are assumed when conflicting compiler options are specified.

Conflicting Compiler Options		Options Assumed	
FIPS	FLAG¬=I	FIPS	FLAG=I
FIPS	LANGLVL(66)	NOFIPS	LANGLVL(66)
LANGLVL(77)	NAME	LANGLVL(77)	Ignore NAME
NOTRMFLG	VEC(REP(TERM...))	NOTRMFLG	Ignore VEC(REP(...))
OPT=(0|1|2)	VEC(LEV(1|2))	OPT=3	VEC(LEV(1|2))
SYM	NODECK and NOOBJ	NOSYM	NODECK and NOOBJ

Modifying Compilation Options-@PROCESS Statement

The options specified when the compiler is invoked remain in force for all source programs being compiled unless they are overridden with the @PROCESS statement.

Each source program requires its own @PROCESS statement if the options specified are to be overridden when the compiler is invoked. If any source program does not have its own @PROCESS statement, it is compiled according to the compiler-invocation specifications, not according to the @PROCESS specifications of the preceding source program in the job stream.

To change the compiler options, place the @PROCESS statement just before the first statement in the source program. The following rules apply:

(1) @PROCESS must appear in columns 1 through 8 of the statement.
(2) The @PROCESS statement can be followed by compiler options in columns 9 through 72 of the statement. The options must be separated by commas or blanks.
(3) Multiple process statements can be supplied for a program unit. Columns 9 through 72 of a following @PROCESS statement are appended to the previous @PROCESS statement. There may be up to 20 @PROCESS statements.

All compiler options except OBJECT and DECK are permissible.

Intervening lines must not appear between @PROCESS statements.

(4) If NODECK or OBJ has been specified in the PAR field of the $RUN command, then DECK or NOOBJ, respectively, cannot be specified on the @PROCESS statement.

(5) TERMINAL and TRMFLG cannot be specified on the @PROCESS statement if TERMINAL was not specified on the EXEC statement or in the system defaults.

THE INCLUDE FEATURE

The INCLUDE feature is supported. For a description of INCLUDE, see the IBM publication, VS FORTRAN Version 2: Language and Library Reference.

The compiler expects to find the user's INCLUDE file in the form of an MTS macro library. A macro library may be built by the public file *MACUTIL (see the description of *MACUTIL in MTS Volume 2, Public File Descriptions).

VS FORTRAN I/O LIBRARY

The file *FORTRANLIB contains all the necessary routines for vectorized programs produced by the VS FORTRAN compiler. This library is also available for all other FORTRAN programs, but only programs produced by Version 2 of VS FORTRAN can fully utilize the IBM 3090 Vector Facility.

The features available in the library are described briefly below:

(1) *FORTRANLIB contains new mathematical functions with improved precision and greater speed. These functions are not compatible with the routines in the Elementary Function Library (EFL). Extended-precision mathematical routines, which are rather slow and expensive, are also included in *FORTRANLIB, but currently FORTRAN I/O does not support I/O conversion for extended-precision numbers.

(2) *FORTRANLIB contains two routines, FCXPC# and FCDCD#, that provide exponentiation of a complex base to a complex power, which is now allowed in the ANSI FORTRAN 77 standard. The Elementary Function Library does not contain these routines.

(3) The new functions, including all IBM mathematical, character, and bit functions as described in Chapter 8 of the IBM publication, VS FORTRAN Version 2: Language and Library Refer-

ence, do not recognize the optional arguments described in the section "The Elementary Function Library" of this Volume 3.

(4) The new functions do not call the MTS EFL module ERRMON#. The IBM error monitor is called instead. Users can take advantage of the IBM extended error-handling subroutines ERRMON, ERRSAV, ERRSET, ERRSTR, and ERRTRA, as described in Chapter 10 of the IBM Language and Library Reference. I/O errors are covered by MTS FORTRAN I/O routines, but not by the IBM error monitor.

(5) The symbolic dump subroutine SDUMP is available in *FORTRANLIB for VS FORTRAN programs. This subroutine is described in the IBM Language and Library Reference and is not the same as the MTS subroutine SDUMP as described in MTS Volume 3, System Subroutine Descriptions. A similar subroutine STDDMP can be used instead of the MTS subroutine SDUMP. *FORTRANLIB also contains two character-dump subroutines, CDUMP and CPDUMP.

(6) For the convenience of FORTRAN users, the free-formatted I/O routines FREAD and FWRITE, are provided in *FORTRANLIB.

It is not necessary for Version 2 VS FORTRAN programs to be concatenated with *FORTRANLIB unless users want to use the new mathematical functions, the IBM SDUMP subroutine, or the IBM extended error-handling subroutines. If the programs were compiled with compiler options VECTOR or IL(NODIM), they must be concatenated with *FORTRANLIB,

```
$RUN object+*FORTRANLIB
```

October 1983

WATFIV

INTRODUCTION

WATFIV and its predecessor WATFOR[1] are FORTRAN-IV compilers developed at the University of Waterloo, Waterloo, Ontario, Canada. These compilers were developed to provide extremely fast translation of student programs. In addition to fast translation, the WATFIV compiler provides an extensive set of both compile-time and execution-time diagnostics to aid in the debugging of programs.

WATFIV is a "compile-and-execute" compiler; this means that source programs are compiled, loaded, and executed by the compiler. This feature minimizes both compiling and loading times for programs. However, since there is no way to obtain an object module, programs must be recompiled each time they are used. In addition, the machine code produced by the compiler is not particularly efficient. Programs that are to be used more than once should only be debugged in WATFIV. The program can then be recompiled using either the FORTRAN-G or FORTRAN-H compiler under *FTN to produce a more efficient object version.

FORTRAN G and H do not support all of the WATFIV features described in this section. Users intending to compile their programs later using another compiler should see the section "Incompatibilities of WATFIV." In addition, the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515, should be consulted.

WATFIV does not use the standard FORTRAN-IV calling sequences for subroutines, thus care must be used in attempting to link other object modules to WATFIV-compiled routines. Users intending to use previously compiled subroutines should note especially the sections "Incompatibilities of WATFIV" and "360-Assembly Language Subprograms." The WATFIV calling conventions essentially require that the subroutines be written in PL/360 or 360-assembly language.

The compiler is invoked by the $RUN command as follows:

```
$RUN *WATFIV [logical unit specifications] [PAR=SIZE=n]
```

where the information within brackets is optional as explained in the following section.

---

[1]WATFOR was supported in MTS from 1968 to 1971; in February 1971, it was replaced by WATFIV.

Logical I/O Unit Specifications

The following logical I/O units are used by WATFIV:

SCARDS - compiler input (control commands and source program) and data for READ and READ n statements (defaults to *SOURCE*).

SPRINT - compiler output (source listing and diagnostic messages), output from PRINT and PRINT n statements, and job-accounting information (defaults to *SINK*).

SPUNCH - output from PUNCH and PUNCH n statements (defaults in batch mode to *PUNCH* if a nonzero card estimate was specified on the $SIGNON command; no default in conversational mode).

SERCOM - error comments and prompting messages in conversational mode (defaults to *MSINK*).

0 - library of subroutines in source or object module form (defaults to the file *WATLIB).

1-19 - input: data for READ (i,n) statements (defaults to the file or device assigned to SCARDS). "i" is the logical I/O unit number and "n" is a format statement number.

1-19 - output: output from WRITE (i,n) statements (defaults to the file or device assigned to SPRINT). "i" is the logical I/O unit number and "n" is a format statement number.

The defaults for the logical units 1-19 are SCARDS on input and SPRINT on output. This means that if SCARDS is assigned to a file, e.g., DATAFILE, on the RUN command, and if logical unit 5 is defaulted,

```
$RUN *WATFIV SCARDS=DATAFILE
```

and a statement of the form

```
READ (5,100) X,Y,Z
```

is executed, then the program will read the data from DATAFILE, not from *SOURCE*. On output, a similar condition exists. For example, if SPRINT is assigned to a file, e.g., -T, on the RUN command, and if logical unit 6 is defaulted,

```
$RUN *WATFIV SPRINT=-T
```

and a statement of the form

```
WRITE (6,200) Z,Q
```

is executed, then the program will write the data to the file -T, not to *SINK*. If any question about defaults arises, the best policy is to explicitly specify the assignments for each individual logical I/O unit if one or more are not being defaulted, e.g.,

```
$RUN *WATFIV SCARDS=DATAFILE SPRINT=-T 5=*SOURCE* 6=*SINK*
```

Any type of file or device may be used for logical units 1-19 with the following restrictions:

(1) If the "ENDFILE n" or "REWIND n" statements are used, logical unit "n" must be assigned to a line file, sequential file, or magnetic tape unit.
(2) If the "BACKSPACE n" statement is used, logical unit "n" must be assigned to a line file or a magnetic tape unit.
(3) If a direct-access statement is used on logical unit "n", the unit must be assigned to a line file.

If unformatted input/output is done on a magnetic tape, the tape must be formatted VS (see the section "Magnetic Tapes" in MTS Volume 19, Tapes and Floppy Disks.

The SIZE Parameter

The "n" in this parameter specification refers to the number of pages (4096 bytes per page) that are used to compile and execute programs. The default size is 10 pages. This should provide efficient compilation for most programs. If the compiler does not have enough space, it will terminate execution with a message indicating the condition. At that point enough information about the size requirements is given to determine a suitable value for "n".

Control Commands

The following control commands may be used with WATFIV. Each control command must begin in column 1.

/COMPILE initiates program compilation. This command must appear before the program to be compiled. Usually, it immediately follows the $RUN *WATFIV command. Compiler options may be included on this command.

/DATA or
/EXECUTE initiates program execution. This command must appear after the last WATFIV statement and before the data records

(if any). Lines following this command are treated as data for the current job. The end of the data is indicated by the next control command or by an end-of-file. This command must be present even if there are no data records.

/STOP terminates the compiler. This is the last command in the compiler input stream. An end-of-file also terminates the compiler.

The following commands provide control over the content and appearance of source listings; they are placed in the source program.

/PRINTOFF terminates the source listing. The /PRINTOFF command itself is not printed.

/PRINTON restarts the source listing if a previous /PRINTOFF command was used or if a NOLIST option was specified on the /COMPILE command. The /PRINTON command is printed.

/EJECT skips to a new page.

/SPACE skips a single line.

The following control commands allow one to change some of the options given on the /COMPILE command. They are placed in the source program.

/WARN prints warning messages from this point.

/NOWARN suppresses warning messages from this point.

/EXT prints extension messages from this point.

/NOEXT suppresses extension messages from this point.

/CHECK initiates checking for undefined variables from this point.

/NOCHECK suppresses checking for undefined variables from this point.

An execution-time trace may be obtained by using the following commands.

/ISNON turns on the tracing of statements by internal statement number (ISN). This control command must be preceded by at least two executable statements. ISN tracing remains on until a /ISNOFF command is encountered.

/ISNOFF turns off ISN tracing.

The /MTS command returns control to MTS. WATFIV processing may be resumed with a $RESTART command. This sequence may be used to reassign logical units or to modify files. For example, at a terminal:

```
WATFIV: ENTER "/COMPILE" OR "/STOP"
User:   /MTS
User:   $RESTART 3=DATA
User:   /COMPILE
WATFIV: ENTER STATEMENTS
```

This reassigns logical unit 3 to the file DATA. The PAUSE statement has the same effect during execution as a /MTS control command during compilation, and, in addition, optionally prints a comment on SERCOM.

The following example illustrates a typical control command and source program structure for WATFIV.

```
$RUN *WATFIV
/COMPILE
     (main program source statements)
/EJECT
     (subroutine SUB1 source statements)
/SPACE
     (subroutine SUB2 source statements)
/DATA
     (data cards)
/COMPILE
     (new main program source statements)
/PRINTOFF
     (subroutine SUB1 source statements)
     (subroutine SUB2 source statements)
/DATA
     (data cards)
/STOP
```

The following examples illustrate the use or nonuse of logical I/O unit specifications with or without control commands.

(1)
```
$RUN *WATFIV
/COMPILE
     (source statements)
/DATA
     (data cards for READ, or READ n, or READ (i,n), where "n"
     is a FORMAT statement number or *)
/STOP
```

(2)
```
$RUN *WATFIV SCARDS=MINE 5=*SOURCE*
     (data cards for READ (5,n), where "n" is a FORMAT statement
     number or *)
$ENDFILE
```

In example (2), the file MINE contains the control statements and source statements. The last command of the file MINE must be /DATA.

(3) $RUN *WATFIV SCARDS=PROGRAM

In example (3), the file PROGRAM contains control commands, source statements, and data for READ or READ n statements.

(4)
```
$RUN *WATFIV 5=DATA
/COMPILE
       .
       .
       .
      READ (5,10) A
10    FORMAT(F15.2)
       .
       .
       .
/DATA
/STOP
```

In example (4), the file DATA contains the data records for the READ (5,10) statement.

(5)
```
$RUN *WATFIV SCARDS=*SOURCE*+DATA+*SOURCE*
/COMPILE
       .
       .
       .
      READ (5,10) A
10    FORMAT(F15.2)
       .
       .
       .
/DATA
$ENDFILE
/STOP
```

Example (5) has the same effect for reading input data as example (4).

(6)
```
$RUN *WATFIV SPRINT=FILEA
       .
       .
       .
```

In example (6), all output is directed to FILEA.

Using Control Commands

WATFIV, like MTS, operates as a command processor. WATFIV expects that all commands will originate from the file or device that SCARDS is either explicitly assigned to or defaulted to. Control commands

appearing in the input stream from SCARDS are detected by the presence of the "/" in column one. Data read from any device other than that assigned to SCARDS will not be scanned for control commands, thus:

```
$RUN *WATFIV 5=*SOURCE*
```

will result in all data read from unit 5 being scanned for control commands, since, in this case, SCARDS=*SOURCE* by default. However,

```
$RUN *WATFIV 5=DATAFILE
```

will not detect control commands in the data stream from unit 5. Hence, control commands and data sets should be structured carefully.

Common Problems Using Control Commands

(1)
```
$RUN *WATFIV
     (program)
/DATA
/STOP
```

This or a similar deck setup will result in an EXECUTION BEGINS and EXECUTION TERMINATED set of messages since there is no /COMPILE command.

(2) Problems often occur with the use of the /STOP command:

```
$RUN *WATFIV
/COMPILE
     (program)
/DATA
     (data)
$ENDFILE
/STOP
```

With the deck set up as shown above, if the program terminates on an end-of-file condition, control will return to WATFIV. WATFIV will then read the /STOP command, terminate execution, and pass control to MTS. This deck setup is subject to another interpretation, however, for if the program terminates without having read the $ENDFILE, this command will be read by WATFIV, and WATFIV will terminate execution. MTS will then read in the /STOP command, flag it as an illegal command, and (in batch mode) proceed to the next legal MTS command.

(3) When a program is in execution under the control of WATFIV and reading data from the device assigned to SCARDS, it may appear that WATFIV control commands are not detected. For example, a program is reading data followed by a /STOP command from *SOURCE* (SCARDS is assigned to *SOURCE*). WATFIV will detect the /STOP command and generate an end-of-file to the program. Should the program not be using the END=parameter on the read statement, no action will be taken. At this point the program

will again attempt to read. WATFIV will forget that it read the /STOP command and the program will read whatever is next as data, unless it is a command, in which case the above sequence is repeated. This is true for all control commands read, not just for /STOP commands. Execution continues in this manner until a program interrupt occurs.

/COMPILE Command Format

Job parameters may be included on the /COMPILE control command. For example,

```
/COMPILE TIME=30,PAGES=10,NOLIST
```

The allowable parameters are described below. Default values will be assumed for parameters that are omitted from the /COMPILE command. Abbreviations, where available, are underlined.

KP={26|29} — 26 specifies that the source was punched on a 026 (BCD) keypunch; 29 specifies that it was punched on a 029 (EBCDIC) keypunch. The default is KP=29.

TIME=m — "m" is an integer or decimal number specifying the maximum number of seconds to be allowed for execution of the program. The default is the time remaining on the MTS time limit (local or global), minus one-fifth of a second.

PAGES=n — "n" is an integer specifying the maximum number of pages to be printed at execution time. The default is the number of pages remaining of the MTS page limit (local or global), minus one page.

LINES=k — "k" is an integer specifying the maximum number of lines to be printed per page. (The compiler uses "k" to provide automatic page-skipping at both compile and execution times.) The default is LINES=60.

CHECK
NOCHECK
RUN=FREE — If CHECK is specified, the compiler will check at execution time for attempted uses of variables which have not been assigned a value (undefined variables). The use of NOCHECK suppresses the check, resulting in somewhat faster execution time and producing somewhat less object code. RUN=FREE is the same as CHECK, but the compiler will initiate execution of the program even if it contained serious source errors. If an executable statement which contained a source error is subsequently encountered, execution is terminated.

See also the /CHECK and /NOCHECK control commands above. The default is CHECK.

LIST/SOURCE
NOLIST/NOSOURCE
: LIST produces a source listing of the program; NOLIST suppresses the listing. See also the /PRINTON and /PRINTOFF control commands. LIST is the default in batch mode; NOLIST is the default in conversational mode. SOURCE is a synonym for LIST.

LIBLIST
NOLIBLIST
: LIBLIST produces a source listing of the subprograms automatically retrieved from a library; NOLIBLIST suppresses the listing of library routines. The default is NOLIBLIST. The LIST/NOLIST and LIBLIST/NOLIBLIST parameters are independent.

WARN
NOWARN
: WARN causes diagnostics to appear in the source listing, and NOWARN suppresses all diagnostics of severity less than a fatal error. The default is WARN. Error severities are discussed in the section, "Diagnostics." See also the /WARN and /NOWARN control commands listed under "Control Commands."

EXT
NOEXT
: EXT causes the extension messages to appear in the source listing, and NOEXT suppresses all extension messages. The default is EXT.

Notes:

(1) Parameters may be entered in any order, e.g.,

```
/COMPILE PAGES=7,NOLIST,TIME=15
```

(2) Parameters are separated by commas and/or blanks and may extend to column 79.

(3) If a parameter is in error, the scan for any remaining parameters is stopped, and default values will be assumed. For example, in

```
/COMPILE PAGES=200,NOWARN,TAME=60,KP=29,RUN=NOCHECK
```

the PAGES=200 and NOWARN parameters are recognized and used by the compiler, but defaults are assumed for all other parameters since the TIME parameter is in error (i.e., in the above example, "TAME" has been mispunched for "TIME".)

(4) If any parameter is specified more than once, the rightmost value is used, e.g.,

```
/COMPILE KP=26,TIME=60,LIST,KP=29
```

results in KP=29 being used.

(5) If the source listing is suppressed (NOLIST) and an error is detected, the first line of the last source statement is printed before the error comment.

Conversational Use of WATFIV

If WATFIV is being run from a terminal, a limited amount of prompting is provided. If the program to be compiled is entered from the terminal, the message

```
ENTER "/COMPILE" OR "/STOP"
```

is printed. If /STOP is entered, WATFIV processing is terminated. If /COMPILE is entered, the message

```
ENTER STATEMENTS
```

is printed. Each statement entered should follow the standard FORTRAN card-column conventions. When the last source statement has been entered, the user must enter

```
/DATA or /EXECUTE
```

to signify the end of his program and initiate execution. If the program is sufficiently error-free, the message,

```
EXECUTION BEGINS ...
```

is printed and execution begins. If there are errors, WATFIV prints

```
EXECUTION SUPPRESSED ...
```

and asks for another /COMPILE or /STOP command.

If WATFIV is being run from a terminal, but the program to be compiled is stored in a file (i.e., SCARDS is assigned to a file), no prompting occurs; however, each WATFIV control command is echoed on the user's terminal.

The job-accounting information and certain other trivial output are not printed when WATFIV is run from a terminal.

Job-Accounting Output

In batch mode, the last few lines of output for each job processed by WATFIV consist of certain accounting information. Specifically, the information provided is:

(1) the time (in seconds) taken to compile the program,

(2) the time (in seconds) for program execution,

(3) the amount (in bytes) of object code[2] generated for the program,

(4) the amount (in bytes) of storage used by the program for arrays, common blocks, and equivalenced variables (the so-called "array area"),

(5) the total storage (in bytes) that was available for the run to contain object code and array area, and

(6) the number of errors, warnings, and extensions issued for the program.

An example of the job-accounting output follows:

```
  CORE USAGE OBJECT CODE =      320 BYTES, ARRAY AREA = 0 BYTES
                                TOTAL AREA AVAILABLE = 40960 BYTES

  DIAGNOSTICS                   NUMBER OF ERRORS = 2,
                                NUMBER OF WARNINGS = 3,
                                NUMBER OF EXTENSIONS = 1

  COMPILE TIME = 0.020 SEC,     EXECUTION TIME = 1.230 SEC,
                                WATFIV - JUL 1973 VIL4
                                16:54:31 TUESDAY 13 MAY 75
```

DIAGNOSTICS

Introduction to Diagnostic Features

WATFIV issues compile-time diagnostic messages at three levels of severity--extension, warning, and error. A diagnostic message is generated in-line in the source listing, immediately below the statement in which the condition was detected.

---

[2]This includes constants, temporaries, nonequivalenced simple variables, save areas, any routines loaded from the object library, etc.

An extension diagnostic message results if an extension of the FORTRAN language allowed by WATFIV is used. These are described in the section, "Extensions." This diagnostic is issued so that the extension may be eliminated before compiling the program with other FORTRAN compilers.

A warning diagnostic message is issued for language violations for which the compiler can take some reasonable corrective action, e.g., truncating a name of more than 6 characters.

An error diagnostic message is issued when a language violation severe enough to prevent execution is encountered. In this case, the compiler will suppress execution of the program, unless RUN=FREE was specified on the /COMPILE command.

At execution time, all errors are fatal[3] in the sense that the compiler will terminate the current job and proceed to the next job (if any) in the input stream. For an execution-time error, the compiler generates a diagnostic and a subprogram traceback in the printed output. This gives the line number of the statement in which the error occurred, the name of the subprogram in which the error occurred, the name of the subprogram which called it, etc., continuing back to the main program (which is referred to as M/PROG). The line number of each statement appears to the left of the statement in the source listing. This line number is compiler-generated, and is distinct from and should not be confused with any FORTRAN statement number the programmer may have assigned to a statement.

Example of a traceback:

```
***ERROR***   VALUE OF A IS UNDEFINED
PROGRAM WAS EXECUTING LINE 15 IN ROUTINE RTN2   WHEN TERMINATION
     OCCURRED
PROGRAM WAS EXECUTING LINE  9 IN ROUTINE RTN1   WHEN TERMINATION
     OCCURRED
PROGRAM WAS EXECUTING LINE  4 IN ROUTINE M/PROG WHEN TERMINATION
     OCCURRED
```

One of the design goals of WATFIV was to supply good diagnostics. The implementors at the University of Waterloo think the goal has been well met, but they have heard that a few users of the compiler at their installation have found some of the diagnostic messages to be vague, obscure, or sarcastic. It is hoped that the following paragraphs will simplify, for the user, the interpretation of some of the error messages which may, at present, be too brief or may contain special words with meanings entirely clear only to the compiler implementors.

---

[3]Exception: If an I/O error occurs and the programmer has specified an ERR return in the affected I/O statement, an error message is given and execution proceeds at the statement specified by the ERR parameter.

The user should be aware that an error in one statement may lead to apparent errors in subsequent statements. Thus, if the first error is corrected, the others will disappear as well on a subsequent compilation. This is particularly true when the first error occurs in a specification statement. The reason for this is that the compiler scans each source statement, column by column from left to right, and usually abandons compilation of a statement when a syntax error is encountered. Thus, correct information in a statement may be ignored if it follows a column that contains an error.

Consider the following program as an example:

```
DIMENSION A(10),B(104+C(10)
C(1)=2
   .
   .
   .
```

Both the first and second statements will be flagged with error messages: the first, since there is no matching parenthesis for the dimension of B; the second, since the compiler, lacking knowledge that C is an array because of the previous error, assumes that the second statement is a definition of a statement function C. (Statement function definitions must have variable names, not constants, as dummy arguments.) The second error will disappear when the first error has been corrected.

Thus, the programmer, when confronted with an error message, must do some analysis to see if it is a real error or merely an apparent error arising from an error in a previous statement.

Certain error messages generated by the compiler rely on the programmer's knowledge of the compilers left-to-right scan of statements. These messages usually relate to the syntax of statements, and contain the word "expecting"; for example, the statement

```
GOTO,
```

is flagged with the message,

```
EXPECTING OPERATOR BUT , BEFORE END-OF-STATEMENT WAS FOUND
```

This implies that the compiler, scanning the statement from left to right, expected to find an operator after the word GOTO in order to consider the statement syntactically correct according to the rules of FORTRAN, but did not find such an operator.

Glossary of Terms

The following glossary defines some terms which appear in the WATFIV diagnostics and which may not have a "standard" or accepted meaning to FORTRAN programmers.

Argument

A value passed to a subprogram. For example, A, 3.5, and SIN(X) are arguments in the following statement:

```
CALL SP1(A,3.5,SIN(X))
```

Assigned GOTO Index

A variable used in an ASSIGN statement or assigned GOTO statement, e.g., I is an assigned GOTO index in the following statement:

```
ASSIGN 5 TO I
```

Defined

At compile time, the mode and/or type of a symbolic name is defined when there is no longer any doubt as to what its mode and/or type might be. The mode and/or type can be established explicitly from information in specification statements which refer to the symbolic name, or implicitly from the first use of the name in a program segment. Once the mode and/or type of a name have been defined, they may not be redefined. Consider the following sequence of statements:

```
REAL I,J(10),K,L*8/1.D0/
DIMENSION I(5)
EXTERNAL K
M=L + FN(I)
```

The first statement defines the type as REAL for all four variable names, I, J, K, L. Furthermore, it defines the modes of names J and L; J is explicitly declared as an array, and L is assumed to be a simple variable since it is initialized. The second and third statements explicitly define the modes of names I and K, as array and subprogram, respectively. The fourth statement implicitly defines the mode and type of names of M and FN since they are used in that statement; since this is their first appearance in the program, their types are determined from the FORTRAN first-letter rule, and their modes are established from their uses; M is a simple integer variable, FN is a REAL*4 function. At execution time, a variable or array element or function name is defined if it has been assigned a value.

Dimension

A value used to declare the maximum value that a subscript of an array may assume at execution time. For example, 10, 15, and 5 are dimensions of A in the following statement:

```
DIMENSION A(10,15,5)
```

DO-Loop Parameter

A simple integer variable or integer constant used to control the number of times a DO-loop is performed. For example: I, 3, J, and 2 are DO-loop parameters in the following statement:

```
DO 17 I=3,J,2
```

End-of-Statement

The implied end-of-statement operator that the compiler expects to find at the end of a correct statement.

FORTRAN Keyword

A word, such as STOP, READ, or GOTO, which identifies a FORTRAN statement.

Mode

This generally refers to the use of a symbolic name within a subprogram, or to a program as a whole. Use means a variable name, common block name, subprogram name, etc. Thus, the name AB has mode "common block" in the statement:

```
COMMON /AB/X,Y,Z
```

Sometimes the mode may include type as well, e.g., the symbolic name FN has mode "REAL*8 function subprogram" in the following example:

```
REAL FUNCTION FN*8(A,B)
```

Object of a DO

The last statement of a DO-loop. The statement numbered 15 is the object of the DO-loop defined by the statement numbered 7 in the following example:

```
7     DO 15 I=2,J,2
      A(I)=I*2
15    X(I)=A(I)*B(I)
```

Operator

This is usually an arithmetic operator such as "+", "-", etc., but it may be any delimiter, e.g., "(", "&", ",".

Parameter

A symbolic value used in a subprogram and replaced by a real argument when the subprogram is referenced at execution time (sometimes called "dummy arguments"). For example, A and B are parameters in the following statement:

```
SUBROUTINE EGGMOR(A,B)
```

Program Segment

A subroutine or function subprogram, or a main program.

Simple Variable

A variable which is not an array.

Statement Number Constant

The number of a statement preceded by an & in the program. &5 is a statement number constant in the following statement:

```
CALL SUBR(X,&5)
```

Subscript

A value used to refer to a member of an array. For example, I, 7, and 3*K+12 are subscripts of A in the following statement:

```
Y=A(I,7,3*K+12)
```

Symbol

A symbolic name, i.e., the name of a variable, array, subprogram, etc.

Temporary

A value which is the result of evaluating an expression. For example, 3.*A+2. is a "temporary" in the following statement:

```
CALL RTN(3.*A+2.)
```

Type

This usually refers to one of the types LOGICAL, INTEGER, REAL, COMPLEX, and (with WATFIV) CHARACTER. However, it may refer to a particular subtype (type with length). For example, the following

statements define X to have type REAL*4, A to have type REAL*8, and Z to have type LOGICAL.

```
REAL X*4, A*8
LOGICAL Z
```

Undefined

At execution time, a variable or array element is said to be undefined if it has not had a value assigned to it. For example, if the statement

```
X=Y
```

were the first statement of a main program, then, at execution time, Y will be undefined, since there is no way it could have had a value assigned to it. WATFIV will check the program at execution time for attempts to use undefined variables unless RUN=NOCHECK is specified on the /COMPILE command.

Notes

(1) The authors of the compiler do not advocate the use of the RUN=FREE facility; it is provided for those programmers who feel it is desirable to obtain some execution-time output, even from a program which may contain serious compile-time errors. Note that some errors are of such a serious nature that execution will be suppressed even if RUN=FREE is specified, e.g., if memory space cannot be allocated to contain arrays declared in the program.

(2) Under RUN=CHECK or RUN=FREE, the compiler will terminate the program if an undefined variable is used in an expression, i.e., if some evaluation is attempted that involves a variable which has not been assigned a value. However, the compiler will allow printing of undefined values without terminating the program. Such values appear on the page as a string of U's. For example, if the statements

```
I=1
K=2
PRINT, I,J,K,
```

were the first to be executed in a program, the line of output produced by the PRINT statement would appear as

```
1 UUUUUUUUUUU    2
```

Note that U's are still printed for undefined variables even under RUN=NOCHECK. RUN=NOCHECK suppresses only the check for attempted use of undefined variables in the evaluation of expressions.

(3) Extension and warning messages may be suppressed from the source listing by specifying NOEXT and NOWARN, respectively, as /COMPILE command parameters. It is a good practice to specify WARN in the initial stages of debugging a program.

(4) The following compiler-generated names appear in some diagnostics.

```
M/PROG - name of the main program
 //    - name of the blank common block
```

LANGUAGE ACCEPTED BY WATFIV

WATFIV attempts to support the language described in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515, subject to the restrictions given below. In addition, WATFIV supports a number of extensions to the language which are described below.

Extensions

The following language extensions, except for (1), (2), (12), (13), are flagged with extension messages. This means that the program is acceptable to WATFIV, but may not compile on other compilers. The messages can be suppressed by use of the NOEXT parameter on the /COMPILE command.

(1) Free-Format I/O

This allows the programmer to perform I/O without reference to a FORMAT statement. For example, the statement

```
PRINT, A,B
```

will print the values of A and B with standard format. For more detailed information, see the section "Free-Format I/O."

(2) CHARACTER Variables

This is a new type of variable which allows the manipulation of data in the form of character strings. As a by-product, in-core formatting of data may be performed. See the section "CHARACTER

Variables" for complete details. A simple example of the use of a CHARACTER variable follows:

```
CHARACTER A*7
   .
   .
   .
A='FINALLY'
   .
   .
   .
```

(3) Multiple-Assignment Statements

Statements of the form

```
v1 = v2 = ...  = vn = expression
```

are allowed, where the "vi" represent variable names or array elements. The effect is the same as the sequence of statements

```
v1 = expression
v2 = v1
   .
   .
   .
vn = v1
```

e.g., A = B(5) = C = 1.5

(4) Expressions in Output Lists

Expressions may be placed in output statements, e.g.,

```
WRITE (6,2) SIN(X)**2,A*X+(B-C)/2
```

The expression may not, however, start with a left parenthesis because the compiler interprets this as an implied DO-loop list item. For example,

```
PRINT, (A+B)/2
```

would result in an error message. However, the equivalent

```
PRINT, +(A+B)/2
```

is acceptable. CHARACTER constants are forms of expressions acceptable in output statements, e.g.,

```
PRINT, 'VALUE OF X=',X
```

(5) Initializing of Blank Common

Variables in blank common may be initialized in DATA or type statements, e.g.,

```
COMMON X
INTEGER X/3/
```

(6) Initializing Common Blocks

Common blocks may be initialized in other than BLOCK DATA subprograms.

(7) Implied DO-Loops in DATA statements

Implied DO-loops are allowed in DATA statements, i.e., a statement of the form

```
DATA (C(I),I=1,5,2)/3*.25/
```

is valid. In fact,

```
DATA (A(I),I=L,M,N)/constant list/
```

is acceptable if L, M, and N have been previously initialized and at least [(M-L)/N]+1 constants are present in the constant list.

(8) Subscripts in Statement Function Definitions

Subscripts may be used on the right-hand side of statement function definitions, e.g.,

```
F(X) = A(I)+X + B(I)
```

(9) Logical, Complex, or Character Subscripts

The real part of a complex value is converted to an integer, and this value is used for indexing into the array. For example, if Z is complex, and A is an array, then A(Z) is equivalent to A(INT(REAL(Z))). For rules and examples of logical and character values as subscripts, see the section "Additional CHARACTER Features."

(10) Transfer Statements as Objects of DO-Loops

A logical IF statement used as the last statement (object) of a DO-loop may contain a GOTO of any form, PAUSE, STOP, RETURN, or arithmetic IF statement. For example,

```
            DO 25 I=1,N
             .
             .
             .
      25    IF (X.EQ.A(I)) RETURN
```

(11) Exceeding the Continuation Card Limit

A statement may be continued over more cards than is allowed by the FORTRAN-compiler continuation card limit. As many cards as needed may be used.

(12) Multiple Statements Per Line

WATFIV allows the programmer to enter more than one statement on a single line. This is particularly useful for programs that are to be stored in libraries since less direct-access storage space is required, and fewer input operations are necessary to retrieve a subprogram.

(a) Only columns 7-72 may be used for statements.
(b) A semicolon is used to indicate the end of a statement.
(c) The normal continuation card rules are used for a statement which is to be continued beyond column 72.
(d) Statement numbers appear in columns 1-5, as usual, or following a semicolon and followed by a colon. A statement number may not be split onto a continuation card.
(e) Comment cards and FORMAT statements must be punched in the conventional manner.

Example:

```
Column 6
      |
25     A=B;C=D;39:PRINT, A,B,
      *C,D;X=A+B*C+D
       PRINT, X; 99:  STOP;END
```

This could be punched in the conventional manner as

```
25    A=B
      C=D
39    PRINT, A,B,C,D
      X=A+B*C+D
      PRINT, X
99    STOP
      END
```

(13) Comments on FORTRAN Statements

The compiler terminates the left-to-right scan of a particular card when a 12-11-0-7-8-9 multipunch is encountered. Effectively, this means comments may follow a FORTRAN statement on the

same line if this multipunch is used to terminate the FORTRAN statement. Note that this card code does not have a graphic symbol assigned to it. It may be punched using the 6-digit multipunch or 3-character multipunch GQZ. Terminal users may use the hexadecimal equivalent "FF" if hexadecimal input editing is enabled.

(14) Additional Debugging Aids

Additional debugging aids have been implemented (see the later section "Debugging Aids").

Free-Format I/O

Free-format I/O is a programming convenience for two reasons:

Inexperienced programmers can defer the use of FORMAT statements until some experience and confidence have been gained in FORTRAN, but can still write programs that involve I/O;

Experienced programmers will find free-format output statements convenient for producing debugging output without having to code associated FORMAT statements.

(1) Source Statement Forms

Free-format I/O has been implemented in WATFIV for use with statements of the following forms:

```
READ, list
PRINT, list
PUNCH, list
READ (unit,*,END=m,ERR=n) list
WRITE (unit,*) list
```

The I/O for the first three forms is done on the standard reader, printer, and punch units, i.e., SCARDS, SPRINT, SPUNCH, respectively. The asterisks in the last two forms imply free-format I/O, and "unit" may be a constant or variable unit number. Like the conventional READ statement, the END and ERR returns are optional. Some examples follow:

```
READ, A,B,(X(I),I=1,N)
PRINT, (J,Z(J),J=N,K,L),I,P
WRITE (6,*) 'DEBUG OUTPUT',99,X,Y,Z+3.5
READ (I,*,END=27) (X(J),J=1,N)
PUNCH, 'ID=',ID,'X=',X
```

(2) Input Data Forms

Data items may be entered one per line, or many per line; in the latter case, data items must be separated by a comma and/or one or more blanks. The first data item on a line need not start in column 1. A data item may not be continued across two lines, i.e., the end of a line acts as a delimiter.

Successive lines are read until enough items have been found to satisfy the requirements of the "list" part of the statement. Any items remaining on the last line read for a particular READ statement will be ignored since the next READ statement executed will cause a new line to be read.

It is valid to use free-format READ statements and conventional READ statements in the same program.

The forms of data items which may be used for various types of FORTRAN variables are:

Integer - signed or unsigned integer constant.

Real - signed or unsigned real constant in F, E, or D forms.

Complex - two real numbers enclosed in parentheses and separated by a comma, e.g., (1.2,-3.8).

Logical - a string of characters containing at least one T or F. The first T or F encountered determines the logical value. If there is no T or F in the character string, it is flagged as an illegal character string.

Character - a string of characters enclosed by quotes. If a quote is required as input, two successive quotes (no blanks between) should be entered. The type of a data item must match the type of the variable it is being read into.

A duplication factor may be used as a shortcut when the user desires to enter the same constant many times. For example, with the statements

```
DIMENSION A(25)
READ, A
```

The data for the READ statement could be entered as

```
15*0.,10*-3.8
```

Examples:

(a)	source statement	READ, X,I,Y,J
	typical data	2.5 3,-7.9, -41
(b)	source statement	COMPLEX Z(5) READ, (Z(I),I=1,3)
	typical data	(5.2,-16.0) 2*(0.,.5E-3)
(c)	source statements	LOGICAL L1,L2,L3 READ, L1,L2,L3
	typical data	T .FALSE. , CAT
(d)	source statements	CHARACTER A*1, B*3 READ, A,B
	typical data	'A','DOG'

(3) Output Forms

The compiler supplies formatting for list items written by free-format statements. Line overflow is automatically accounted for, i.e., several records may result from one output statement.

The formats used are:

Integer	- I12
Real*4	- E16.7
Real*8	- D28.16
Complex*8	- '(' E16.7 ',' E16.7 ')'
Complex*16	- '(' D28.16 ',' D28.16 ')'
Logical	- L8
Character*n	- An

CHARACTER Variables

At a meeting held during the SHARE XXVIII Conference in San Francisco in February 1967, the SHARE FORTRAN Project proposed that IBM adopt a new type of variable as an extension to the FORTRAN language supported by the IBM compilers. The following material was adapted from Appendix

B of the Minutes of that meeting since it defines, for the most part, WATFIV's implementation of CHARACTER variables.

Character data are recognized as a legitimate data form which may be manipulated to a limited extent. The general effect to the language is:

CHARACTER is a variable type.

Core-to-core READ and WRITE statements allow core-to-core formatting.

Implicit record-size for CHARACTER arrays for FORMAT statement control is defined in the TYPE statement (not in the READ and WRITE statements).

A WRITE statement may be used to define a variable.

A variable of type CHARACTER represents a character string (literal data). The standard (default) length of a character string is 1; WATFIV permits strings of up to 255 characters to be defined. A programmer may declare a variable to be of type CHARACTER by use of an IMPLICIT statement or a CHARACTER statement.

(1) IMPLICIT Statement

The type CHARACTER is permitted in the IMPLICIT statement with a specified length. If the length is omitted, the standard length of 1 is assumed. For example,

```
IMPLICIT CHARACTER*80 (A-D), CHARACTER ($,Z)
```

This example declares all variables beginning with the characters A through D as CHARACTER type, with each variable or array element 80 characters in size. All variables beginning with the characters $ and Z are also declared as CHARACTER. Since no length specification was explicitly given, 1 character (the standard length for CHARACTER) is allocated for each variable.

(2) CHARACTER Statement: General Form

The general form of the character statement is

```
CHARACTER*s a*s1(k1)/x1/,b*s2(k2)/x2/,...,z*sn(kn)/xn/
```

where *s,*s1,*s2,...,*sn (optional) are the character string lengths, each between 1 and 255; a,b,...,z are the variable and/or array names; (k1),(k2),...,(kn) (optional) are the array dimensions,each composed of 1 to 7 unsigned integer constants separated by commas, exactly as for integer and real arrays. In a subprogram, unsigned integer variables are also permitted. /x1/,/x2/,.../xn/ (optional) are initial data values; each is a list of constants separated by commas.

The dimension information may be included in the CHARACTER statement, or may be placed in DIMENSION or COMMON statements.

Initial data values may be assigned to variables or arrays by use of /x/, where "x" is a constant or list of constants separated by commas. This set of constants may be in the form "r*constant", where "r" is an unsigned integer, called the repeat constant. The initial data values may only be literal constants and must be the same length as or shorter than the corresponding variable or array element. Initial data values will be truncated from the right (and diagnosed) if too long, and they will be padded with blanks on the right if too short (see example 2 below).

Initial data values may be given for a variable or array in blank or labeled common.

The CHARACTER statement overrides the IMPLICIT statement. If the length specification (i.e., *s) is omitted, the standard length of 1 is assumed. If an array is used as a parameter to a subprogram and is not in a COMMON block, the size of this array may be specified implicitly by an integer variable of length 4 which can appear explicitly in the SUBROUTINE statement or implicitly in COMMON (adjustable dimensions). In this respect, character arrays behave in exactly the same way as arrays of other types.

Example 1:

```
CHARACTER*80 CARDS(10),LINES*132(56,2),TCARD
```

This statement declares that the variable TCARD and the arrays named CARDS and LINES are of type CHARACTER. In addition, it declares the size of the array CARDS to be 10 and array LINES to be 112 (2 groups of 56 each). Each element of the array LINES is assigned 132 characters for a total of 14,784 (112 times 132) for the array. Each element of the array CARDS and the variable TCARD is assigned 80 characters (the length associated with the type). The array CARDS is assigned a total of 800 characters.

Example 2:

```
CHARACTER X*3(4)/'ABC','DEFG','HI',JKL'/
```

This statement declares that the array X of four elements of three characters each has initial values:

```
X(1)        ABC
X(2)        DEF
X(3)        HI
X(4)        JKL
```

The statement is incorrectly written, and the value specified for X(2) has been altered by truncating the character G. The value specified for X(3) has been padded with a blank on the right.

(3) Character Variables in Other FORTRAN Statements

CHARACTER type variables and array names may appear in the following statement types:

```
DIMENSION
COMMON
NAMELIST
CALL
SUBROUTINE
FUNCTION
```

DATA statement: CHARACTER variables, array element names, or array names may appear in DATA statements. The data values may only be literal constants and must be the same length as, or shorter than, the corresponding variable or array element. Initial data values will be truncated from the right and diagnosed if too long, or padded with blanks on the right if too short (see example 2 above).

EQUIVALENCE statement: CHARACTER variables, arrays, or array elements may appear in EQUIVALENCE statements. CHARACTER data may be equivalenced to other than CHARACTER data, but the equivalence implies storage sharing only. Consider the example:

```
CHARACTER A*5,B*2,C*1
CHARACTER D*1(5)
EQUIVALENCE (D(1),A),(D(2),B),(D(5),C)
   .
   .
   .
```

These statements cause the following alignment of characters:

```
A-----
B --
C    -
```

C and B are thus equivalenced to characters in the middle of A.

FUNCTION reference: CHARACTER variable names, array element names, array names, and literal constants may appear as parameters in a function reference.

Example:

```
      CHARACTER CARD*80
2     READ (6,1) CARD
```

```
1     FORMAT (A80)
      IF (COMPAR(CARD,'END ')) 2,3,2
3     STOP
      END

      FUNCTION COMPAR(STR1,STR2)
      CHARACTER*80 STR1
      CHARACTER*4 STR2
      COMPAR=1
      IF(STR1.EQ.STR2) COMPAR=0
      RETURN
      END
```

An 80-character image is read into the element CARD. The function COMPAR compares CARD with 'END ' and returns a positive or zero numeric value which is used conditionally to terminate the program.

In comparisons with unequal length operands, the shorter operand is considered to be padded on the right with blanks to match the length of the longer operand.

Statement function statements: Nonsubscripted CHARACTER variable names may appear as parameters in a statement function statement.

(4) Core-to-Core Input/Output Statements

An additional type of I/O statement provides for core-to-core transmission of data under FORMAT control. There are two core-to-core I/O statements: READ and WRITE. In a core-to-core operation, no actual input/output takes place; data conversion and transmission take place between an internal buffer and the elements specified by a list.

(a) WRITE statement: The WRITE statement has the general form

WRITE (a,b) list

where "a" is a character array, array element, or variable name which specifies the starting location of the internal buffer to which data is to be transmitted; "b" is a statement number of a FORMAT statement or an array name or array element indicating the beginning location of a format statement which describes the data to be transmitted; and "list" is a series of variable or array names (which may be indexed and incremented) separated by commas. They specify the number of items to be written and the locations in storage from which the data are to be taken.

This form of the WRITE statement causes the data items specified by the list to be converted to character strings, according to the FORMAT specified by "b", and placed in

storage beginning at the first character element specified by "a".

Characters are placed in consecutive character positions in the buffer, starting with the first character position of the first element specified by "a". When a new record is begun, it starts at the first character position of the next element.

The number of characters generated for a record (as specified by the FORMAT statement and "list") should not be greater than the size of the element specified by "a". If fewer characters are generated than necessary to fill the element, it is filled with trailing blanks.

Example 1:

```
          CHARACTER M*12
           .
           .
           .
          I=15
          J=7
           .
           .
           .
          WRITE (M,2) I,J
    2     FORMAT (2H(F,I2,1H.,I1,1H))
           .
           .
           .
```

These statements might be used to create, for later use, a format stored in variable M. The format so created would appear as:

```
     (F15.7)bbbbb
```

where "b" represents the character blank.

Example 2:

```
          CHARACTER M*12,N*132
           .
           .
           .
          K=FUNC(A,B,C,D)
           .
           .
           .
    2     WRITE (M,4) K
    4     FORMAT (1H(,I3,6HX,1H*))
    6     WRITE (N,M)
```

```
      .
      .
      .
```

Statement 2 creates a format stored in variable M, which for a value of K of 96, would appear as:

```
(b96X,1H*)bb
```

Statement 6 then uses the above format (in the variable M) to prepare a character string 132 characters long in the variable N which consists of all blanks except for an asterisk in the ninety-seventh character.

(b) READ statement: The READ statement has the general form

```
READ (a,b) list
```

where "a" is a character array, array element, or variable name which specifies the starting location of the internal buffer from which data are to be transmitted; "b" is either the statement number of a FORMAT statement or a character array element indicating the beginning location of a FORMAT statement which describes the data to be transmitted; "list" is a series of array names (which may be indexed and incremented) and/or variables, separated by commas. This specifies the number of items to be read and the locations in storage into which the data are placed.

This form of the READ statement causes the character string beginning at the first character element specified by "a" to be converted according to the FORMAT specified by "b", and stored in the elements specified by "list".

Characters are obtained from the buffer starting with the first character position of the first element specified by "a", from consecutive character positions. When a new record is begun, it starts at the first character position of the next element.

The FORMAT statement and "list" should not require more characters from an element than the length of that element. A new record is begun whenever specified in the FORMAT.

Example:

```
           CHARACTER*80 R(10)
            .
            .
            .
           DO 20 I=1,10
     3     READ (R(I),5) J
     5     FORMAT (I1)
```

```
      GOTO (11,12,13,14,15,16,17,18,19), J
11    READ (R(I),21) (A(K),K=1,10)
21    FORMAT (1X,10F8.3)
      GOTO 31
12    READ (R(I),22) K1,K2,K3,K4
22    FORMAT (1X,4I5)
      GOTO 32
13    READ (R(I),23) X,Y,Z
23    FORMAT (1X,3E20.9)
       .
       .
       .
20    CONTINUE
```

The statements illustrate a method of processing randomly ordered input lines of varying format and data content. The line type is identified by a digit from one to nine in the first column. Statement 3 converts the digit from character form to integer form. The GOTO then transfers to the READ/FORMAT combination which processes the specified format.

(5) Input/Output List

Character variable names, array element names, and array names may appear in input/output lists.

(6) Replacement Statement: A=B

A replacement statement in which all variables, constants, or array elements are of type CHARACTER is permissible. In such a statement the item on the left-hand side may only be a character variable name or a character array element; the item on the right-hand side may be a character variable name, a character array element, or a character (literal) constant.

The element on the right-hand side must be the same length as, or shorter in length than, the element on the left-hand side. The value of the right-hand element will be truncated from the right during replacement and diagnosed if too long, or padded with blanks on the right if too short.

Additional CHARACTER Features

The features of CHARACTER variables described in the following paragraphs were not described in the discussion of the SHARE proposal above, and hence are extensions to the proposal.

It should also be mentioned that WATFIV supplies no particular alignment for CHARACTER variables, unless, of course, they are forced to

some halfword, fullword, or doubleword boundary by COMMON and/or EQUIVALENCE statements.

(1) Use of Subscripts

Subscripts may be of LOGICAL or CHARACTER value. The first character (leftmost byte) in the quantity is used as the low-order byte of a four-byte integer to form the actual subscript. For example, A('123') is the same as A(241) since the internal representation of the character 1, taken as a integer value, is equivalent to 241.

For example,

```
      CHARACTER*1 TRANSL(255),CARD(80)
       .
       .
       .
      DO 1 I=1,80
1     CARD(I)=TRANSL(CARD(I))
       .
       .
       .
```

The above loop will translate each character of a line according to the table TRANSL.

(2) Use with Relational Operators

CHARACTER variables may be used as operands of relational operators provided both operands are of type CHARACTER. All values are treated as if they were in IBM 360 EBCDIC representation. For example,

```
      CHARACTER A*1,B*5,C*5(10)
       .
       .
       .
      IF (A.EQ.C(I)) GOTO 10
       .
       .
       .
      IF (B.LE.'AAAAA') GOTO 30
       .
       .
       .
```

For the purposes of the comparison, when operands of unequal length are involved, the shorter operand is considered to be padded with blanks so that it will be equal to the length of the longer operand. A warning message is issued at compile time when operands of differing lengths are used.

Note that this feature is highly dependent on the IBM 360/370 machine representation of EBCDIC characters.

Restrictions

The user of WATFIV should note the following restrictions in language and facilities provided by the compiler.

(1) The name of a COMMON block must be unique; i.e., it may not also be used as the name of a variable, array, or statement function. This is in violation of the specifications given in IBM System/360 and System/370 FORTRAN-IV Language, form GC28-6515.

(2) The concept of the extended range of a DO-loop defined in GC28-6515 is not supported.

(3) The service subprograms DUMP and PDUMP defined in Appendix C of GC28-6515 are not supported.

(4) The debug facility described in Appendix E of GC28-6515 is not supported.

(5) There are no facilities in WATFIV which correspond to the FORTRAN G/H options MAP, EDIT, XREF, OPT=, DECK, LOAD, NAME=, LIST.

(6) The extended error message facility is not supported.

(7) No overlay facility is available; no "module map" is produced.

(8) The FORTRAN direct-access statements work as described in GC28-6515 with the following exceptions:

   (a) The maximum record length is 247 bytes.
   (b) The DEFINE FILE statement is optional. If it is used, then the "relative position" of a record in the file as used in GC28-6515 is the line number of that record. In this case, only integral line numbers will be used. If the DEFINE FILE statement is not used, then the "relative position" expression in a direct-access READ, FIND, or WRITE statement is taken as the internal form of an MTS line number.

(9) No more than 255 DO statements are allowed in a program segment.

(10) FORMAT is a reserved character sequence when used as the first 6 characters of a statement. It is the only reserved character sequence. For example,

```
FORMAT(I) = 3.5
```

will result in FORMAT error messages, whereas,

```
X = FORMAT(I)
```

is legal, assuming FORMAT to be an array or function name.

(11) WATFIV is a "one-pass" compiler, and requires several restrictions on statement ordering. These are:

(a) Specification statements referring to variables used in NAMELIST or DEFINE FILE statements must precede the NAMELIST or DEFINE FILE statements.

(b) COMMON or EQUIVALENCE statements referring to variables used in DATA or initializing type statements must precede the DATA or initializing type statements. For example,

```
REAL I/5.2/
COMMON I
```

will produce error messages, whereas,

```
COMMON I
REAL I/5.2/
```

is acceptable.

(c) A variable may appear in a EQUIVALENCE statement and then in a subsequent explicit type statement only if the type statement does not declare the length of the variable to be different than could be assumed for it. This assumption is based on the first letter of the variable name, at the time of its appearance in the EQUIVALENCE statement. For example,

```
EQUIVALENCE (A,B)
REAL*8 B
```

will produce an error message, whereas,

```
REAL*8 B
EQUIVALENCE (A,B)
```

will not. Note that

```
EQUIVALENCE (A,B)
INTEGER B
```

is acceptable since the length of B is not changed by the type statement.

(12) Only the following characters are allowed as carriage-control characters on SPRINT. All other characters will be replaced by a blank.

blank	skip 1 line before printing
0	skip 2 lines before printing
-	skip 3 lines before printing
+	no skip before printing
1	skip to first line of next page
2	skip to next 1/2 page
4	skip to next 1/4 page
6	skip to next 1/6 page
8	Skip to logical bottom of page (line 63)
9	suppress space and overflow (i.e., ignore the top and bottom margins of the page)
&	suppress carriage return after printing

Debugging Aids

Some new debugging aids are available in WATFIV. They are the DUMPLIST statement, the ON ERROR GOTO statement, and a statement trace facility.

(1) The DUMPLIST statement is designed especially as a program debugging aid; it is used as follows:

(a) A DUMPLIST statement is essentially a NAMELIST statement, except that the word DUMPLIST replaces the word NAMELIST. The usual rules for NAMELIST statements apply. Sample statements are:

```
DUMPLIST /XXX/A,XYZ,APE/LOK/XX,NEXT
DUMPLIST /THIS/N,TWO,SIX,OLD
```

(b) A DUMPLIST list name need never appear in a READ or WRITE statement.

(c) A DUMPLIST statement has no effect unless the program in which it appears is terminated because of an error condition; then, WATFIV will automatically generate NAMELIST-like output of all DUMPLIST lists appearing in program segments which have been entered. The values printed are those which the variables had when the program was terminated. To avoid producing too much output, only a few key variables should be placed in DUMPLIST statements.

(2) The ON ERROR GOTO statement allows a program which has an error to recover and to take some alternate and possibly corrective action, such as giving a diagnosis. The error exit will be taken only for the first error encountered. The second error will be fatal (to prevent infinite loops). There is a separate error exit for each program segment (main program or subroutine). The last ON ERROR GOTO statement in a program segment determines the error exit location for that program segment. If an error occurs within a program segment for which no error exit

is set up (i.e., no ON ERROR GOTO statement was given in that program segment), the error is fatal. The ON ERROR GOTO statement is not an executable statement; hence, it may be placed anywhere within the program segment it applies to. It is not advisable to have the error exit (GOTO portion of the ON ERROR GOTO) transfer into the range of a DO-loop, as no checking is performed to ensure loop variables are set up properly. Thus, infinite looping may result.

INCOMPATIBILITIES OF WATFIV

By and large, most programs which follow the rules and conventions given in the section "Language Accepted by WATFIV" will produce virtually the same results when run under any of the four compilers: WATFIV, WATFOR, FORTRAN G, or FORTRAN H. However, there will be some programs which will produce different results when run under WATFOR and WATFIV. The difference could be as minor as, for example, an extra warning message issued by WATFIV if a specification statement follows an executable statement. These differences arise because of slightly different conventions used in WATFIV.

A similar situation exists for the FORTRAN G and FORTRAN H compilers. In this case, the differences arise mainly because our interpretations of some vague sections of the IBM FORTRAN publication, IBM System/360 and System/370 FORTRAN IV Language, are different than those of the implementors of the IBM compilers. Again, it should be mentioned that the differences are fairly minor.

To assist the programmer, this section provides information on currently known incompatibilities. Additional information (in the form of MTS Manual updates) may be distributed from time to time as more incompatibilities are discovered.

Incompatibilities with WATFOR

Since WATFOR, the precursor of WATFIV, has not been available in MTS since 1971, this section will be of use to only a few users.

The most likely cause of difficulty is the use of arrays as subprogram arguments, as discussed in number (11) below. However, the more straightforward incompatibilities are discussed first.

(1) WATFOR does not support the NAMELIST, direct-access I/O, and CHARACTER variable language features.

(2) WATFOR does not support the LIST/NOLIST, LIBLIST/NOLIBLIST, WARN/NOWARN options on the /COMPILE control command.

(3) WATFIV issues warnings if the proper ordering of statements is not followed. The proper order is specifications statements before statement function definitions before executable statements.

(4) With WATFIV, DO-loops may be nested to any depth.

(5) An INTEGER*2 variable may not be used as a unit number in an I/O statement with WATFIV.

(6) WATFOR does not accept source statements in compressed form, i.e., more than one statement per line.

(7) With WATFIV, if the index of a computed GOTO is negative or zero, control transfers to the next executable statement; this follows the specifications of GC28-6515. Under WATFOR, a terminating error message is given.

(8) WATFOR gives special treatment to the $ in IMPLICIT statements. WATFIV assumes that the dollar-sign follows Z in alphabetical order; this is the convention of GC28-6515.

(9) If a function subprogram has additional entry points, WATFOR does not equivalence the variables which are the names of the function and its entry points. WATFIV does this, as prescribed by GC28-6515. For example,

```
FUNCTION A
   .
   .
   .
ENTRY B
   .
   .
   .
B=4
RETURN
```

returns 4 as the value of the function in WATFIV.

(10) The conventions, used by WATFIV, for intermixing EBCDIC and BCD characters in source programs are slightly different than those used by WATFOR.

(a) WATFIV does not allow intermixing of the EBCDIC and BCD quote marks in a program.
(b) If KP=26 is specified, WATFIV uses "$" to denote a statement number argument; WATFOR uses a 12-8-6 multi-punch (EBCDIC "+") for this.

(11) WATFIV gives a different treatment to arrays passed to subroutines as parameters.

(a) WATFIV allows the actual argument to be an array element or a simple variable.
(b) WATFIV uses the dimensions declared for the dummy array in the called subprogram. This ensures compatibility with FORTRAN G and H, and object-time dimensions work as specified by GC28-6515.

Under WATFOR, the dimensions for a dummy array are ignored at execution time. When an array is passed from subprogram to subprogram, the dimensions that are declared for it in the program segment in which it is actually allocated storage are passed as well. These dimensions are then used for subscript calculations.

In addition, under WATFOR and WATFIV, the results will be different if the dimensions of the dummy array differ from those of the actual array passed (see point (b) above).

Incompatibilities with FORTRAN G and H

The differences listed below do not include the language extensions and restrictions given in the section "Language Accepted by WATFIV," nor do they include differences which arise either because object programs compiled under FORTRAN G and H are freely allowed to violate the language rules defined by GC28-6515 (e.g., passing an argument of type INTEGER to the SQRT subroutine), or because the FORTRAN-G and -H compilers accept syntax not defined in GC28-6515. The major causes of differences between WATFIV and FORTRAN G and H are likely to be the treatment of FORTRAN-supplied functions and number conversions.

(1) WATFIV provides execution-time page skipping, controlled by the LINES job parameter to the /COMPILE control command.

(2) WATFIV allows any number of contiguous comment lines; comment lines may precede a continuation line. For example,

```
      INTEGER A(2
C     THIS IS A COMMENT
     *),BC
```

(3) WATFIV uses only the high-order byte of a logical quantity in logical operations. For example, if A and B are of type LOGICAL*4, execution of the statement

```
A = B
```

causes only one byte to be moved.

(4) DO-loops may be nested to any depth in WATFIV.

(5) WATFIV supports both EBCDIC and BCD "+" as a carriage-control character.

(6) WATFIV considers the program to be in error if it executes a "RETURN i" statement in which the value of "i" is undefined, zero, negative, or greater than the number of statement-number arguments which appeared in the argument list of the CALL statement that invoked the subprogram.

(7) WATFIV prints no message equivalent to the IHC210I ("OLD PSW IS ...") message when an interrupt occurs.

(8) With WATFIV, the use of a T format, which does a backward tab in an output buffer, does not cause existing characters in the buffer to be blanked out. For example, consider the statements:

```
          K=9
          J=1
          WRITE (6,7) K,J
7         FORMAT ('$$$.00',T3,I2,T6,I2)
```

With WATFIV, the line appears as:

```
$$9.01
```

With FORTRAN G or H, it appears as:

```
$ 9. 1
```

Actually, this is a consequence of the fact that the WATFIV formatting routines assume the buffer to be blanked before any filling occurs, i.e., only significant characters are moved into the buffer.

(9) REAL*4 values are printed with a maximum of 7 significant digits. If the output format specification calls for more, i.e., E20.10, zeros are supplied on the right.

(10) WATFIV treats FORTRAN-supplied functions differently than FORTRAN G and H as follows:

(a) The function type must be explicitly declared if it is different than can be assumed from the implicit rules.
(b) WATFIV makes no distinction between in-line and out-of-line functions; all functions are out-of-line.
(c) WATFIV evaluates all functions that require complicated approximation formulae in double precision, i.e.,

```
SQRT(X)
```

is calculated as, essentially,

```
SNGL(DSQRT(DBLE(X))).
```

(11) WATFIV handles FORMAT statements differently than FORTRAN G and H as follows:

   (a) Commas are not required between format codes in WATFIV; however, a warning message is issued when a comma is not provided.
   (b) WATFIV allows an arbitrary number of continuation cards for FORMAT statements.
   (c) WATFIV does not allow group or field counts to be zero.

(12) Execution-time data lines read on SCARDS by WATFIV-compiled programs may not contain a $ or / in column 1.

(13) WATFIV treats a floating-point constant of more than 6 or 7 significant digits as a double-precision constant. In FORTRAN G and H, one must denote this with a D at the end to get double precision. A constant of this type will cause problems when used with a REAL*4 constant in DATA statements. A message will be generated that data type and constant type do not match.

(14) WATFIV will not accept the *FTN extension that input data fields can be compressed and separated by commas. Thus, all data to be read by a format must be entered according to that format, even at a terminal.

(15) There is a restriction in FORTRAN that the dummy argument for an ENTRY point must not be used in an executable statement prior to the ENTRY point unless it has been previously defined as a dummy argument in an ENTRY, SUBROUTINE, or FUNCTION statement. However, WATFIV regards type declarations as "executable" statements; these will be flagged as errors when they declare type for dummy argument of an ENTRY point.

SUBPROGRAM FACILITIES

This section provides information on the subprogram facilities available with the WATFIV compiler. Rules for passing values between subprograms are also discussed.

Sources of Subprograms

Any subprograms referenced in a FORTRAN program run under WATFIV must come from one of three possible sources:

(1) source programs or object modules in the input stream (SCARDS), i.e., the usual program input;

(2) resident library routines internal to the compiler itself. For example, the routines EXP, DEXP, ALOG, ALOG10, DLOG, DLOG10, EXIT, and SQRT are always resident in memory;

(3) Routines from *WATLIB or other subroutine library files.

The search for subprograms is made in the order given above, i.e., the user may supply a subprogram EXIT which will be used in preference to subprograms which may be resident in memory or in a subroutine library file.

Normally, a user need not be concerned with determining which routines are resident in memory.

FORTRAN-Supplied Routines

The WATFIV user has available all function and subroutine subprograms (except DUMP and PDUMP) mentioned in Appendix C of IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515. The coding used for the double-precision versions of the mathematical functions is essentially that used with the IBM FORTRAN library (without the extended error message facility). Consequently, the algorithms used and error estimates for these routines may be found in the IBM publication, IBM System/360 FORTRAN IV Library Subprograms, form GC28-6596.

The following additional points should be noted. Single-precision versions of many of the mathematical functions used in WATFIV produce the truncated value of the corresponding double-precision version. (Exceptions are the functions such as ABS, MOD, FLOAT, etc., which do not require complicated approximation formulae.) For example, the evaluation of SQRT by WATFIV is essentially equivalent to

```
SQRT(X)=SNGL(DSQRT(DBLE(X)))
```

WATFIV supplies no automatic declarations of FORTRAN mathematical functions. Thus, a user must explicitly declare any FORTRAN function names which are not declared by implicit rules, e.g.,

```
REAL*8 DSIN,DSQRT
COMPLEX CMPLX,CDSIN*16
```

Subprogram Arguments

The rules for passing values between subprograms are generally the same as those described in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515. The relevant sections in that manual are "Dummy Arguments in a Function or Subroutine Subprogram," "Multiple Entry into a Subprogram," and "Object-Time Dimensions." The following remarks augment the rules stated in GC28-6515.

(1) Dummy Arguments

If a dummy argument of a called subprogram is an array, then GC28-6515 specifies that the corresponding actual argument provided by a calling routine must be (1) an array name, or (2) an array element. Furthermore, in case (1), the size of the dummy array as declared in the called subprogram must not exceed the size of the actual array provided by the calling subprogram. (Here "size" means amount, in bytes, of memory allocated.) In case (2), the size of the dummy array must not exceed the size of that portion of the actual array, which follows and includes the specified element.

WATFIV allows a third possibility; namely, that the actual argument may be a simple variable (or expression). The rule is similar to that of case (1); the size of the dummy array must not exceed the number of bytes occupied by the simple variable.

All three rules can be stated more briefly, if somewhat less precisely, by a single rule: the dummy array must fit into the space provided by the actual argument, i.e., the dummy array may be smaller, but may not be larger. These rules are in the language presumably so that programmers will not index beyond the confines of an array, thus possibly destroying other data or program areas. WATFIV ensures that the rules are not violated at execution time by making checks on arguments that are passed to dummy arrays. If a rule is violated, the program is presumed to be at fault, and is terminated with an error message and a subprogram traceback.

An example of case (2) follows in which the dummy array is smaller than the actual array. Note that, according to the rules, B could be dimensioned at, but not larger than, 76.

```
     DIMENSION A(100)
        .
        .
        .
     CALL RTN(A(25))
        .
        .
```

```
      .
      END

      SUBROUTINE RTN(B)
      DIMENSION B(50)
         .
         .
         .
      END
```

Object-time dimensions can be very useful for creating subprograms, especially when it is not known beforehand what dimensions should be used for dummy arrays. The following example illustrates this point.

```
      DIMENSION A(100)
         .
         .
         .
      CALL RTN(A(25),76)
         .
         .
         .
      CALL RTN(A(I),101-I)
         .
         .
         .
      END

      SUBROUTINE RTN(B,N)
      DIMENSION B(N)
         .
         .
         .
      END
```

(2) Hollerith Constants

The following remarks pertain to the use of Hollerith (or CHARACTER) constants as subprogram arguments. Since CHARACTER variables are implemented in WATFIV, a Hollerith constant should be passed to a dummy argument that is a CHARACTER variable of appropriate length. This is merely an application of the general rule that an actual argument should agree in type and length with its corresponding dummy argument. An example follows.

```
         .
         .
         .
      CALL RTN('LENGTH1S9')
         .
         .
```

```
      .
      END

      SUBROUTINE RTN(X)
      CHARACTER*9 X
         .
         .
         .
```

However, to allow some compatibility with existing programs, Hollerith constants used as subprogram arguments are also treated in the following way. The compiler pads the constant on the right, with blanks, to make its length a multiple of four, if necessary. It is then treated as a REAL or INTEGER vector, with a dimension equal to the number of fullwords the constant occupies. Thus, the corresponding dummy argument must be a REAL or INTEGER vector of appropriate dimension. The following example illustrates this.

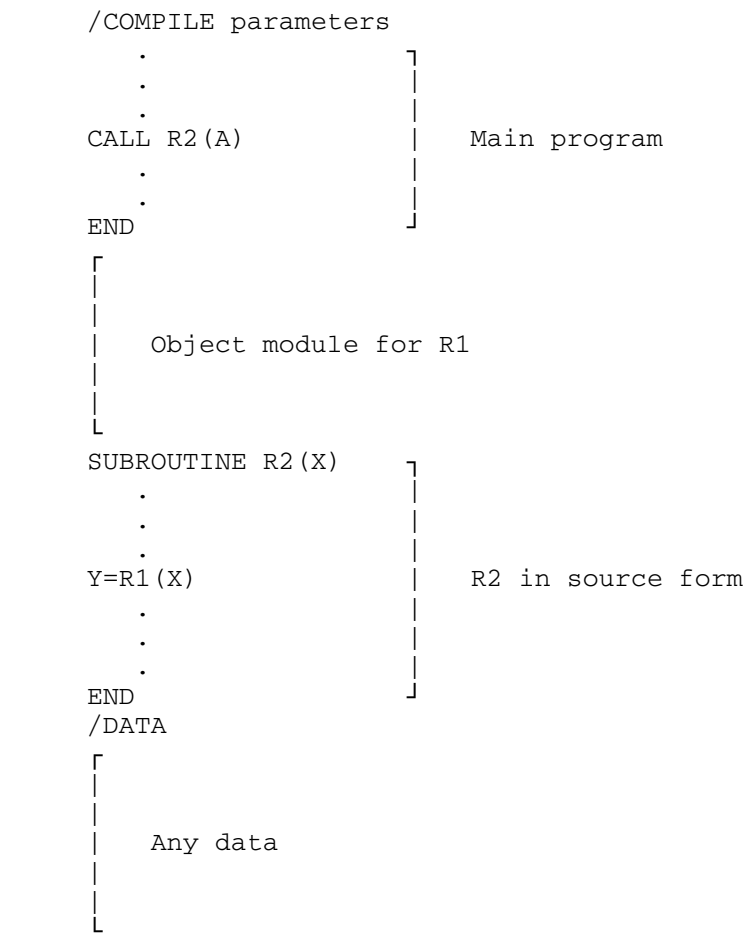
```
         .
         .
         .
      CALL RTN('LENGTH1S9',3)
         .
         .
         .
      END

      SUBROUTINE RTN(I,N)
      DIMENSION I(N)
         .
         .
         .
```

Hollerith constants are always aligned on a fullword boundary.

Subprograms in Object-Module Form

WATFIV will accept subprograms in object-module form from the input stream (SCARDS).

A subprogram in object-module form may appear in any place that a subprogram in source form may appear, but object modules are never listed. The example below shows a job composed of a main program and two subprograms, R1 and R2, in object-module and source form, respectively.



The question naturally arises, "May object modules acquired from the FORTRAN G or H compilers be used?" The answer is, "Yes, but only under certain circumstances." However, the circumstances are so restrictive that, effectively, the answer is, "No." The intention is that the object-module loading facility of WATFIV will be used with special-purpose routines, e.g., plotter routines or 360-assembly language routines.

Since the calling-sequence conventions are similar to those used with the FORTRAN G or H compiler, anyone who previously has coded 360-assembly subroutines should have little difficulty adapting the subprograms for use with WATFIV. Complete details can be found in the section, "360-Assembly Language Subprograms."

(1) An object module is detected by the 12-2-9 punch or X'02' that appears in the first column of a record. This punch is usually put there by most assemblers and compilers. Input records without this identifying characteristic are not considered to be a part of an object module.

(2) WATFIV uses an internal loader. This loader is rather simple and can handle only the following types of loader records:

ESD
TXT
RLD
END

Any other records, such as SYM records, put out under the test version of the FORTRAN-G compiler, or REP records are ignored.

(3) The above loader records are expected to be 80 bytes long. Therefore, WATFIV will not accept output from the MTS linkage editor unless specification was for 80-byte records.

Additional Subprograms Supported

WATFIV supports the four function subprograms described in the following:

Function Name	Purpose	Number of Arguments	Type of Arguments	Type of Result
AND[1]	Logical 'and' of arguments	2 or more	Word length[2]	REAL*4
OR[1]	Logical 'or' of arguments	2 or more	Word length	REAL*4
EOR	Exclusive 'or' of arguments	2 or more	Word length	REAL*4
COMPL	Logical 1's complement of argument	1	Word length	REAL*4

[1]The functions AND and OR have alternate entry points LAND and LOR, respectively, available in *WATLIB.
[2]The term "word length" refers to any type of variable that occupies four bytes; e.g., INTEGER*4, REAL*4, LOGICAL*4, CHARACTER*4, etc. All 32 bits of each argument are used in composing the result of the function evaluation.

Structure of a Subroutine Library

A WATFIV subroutine library consists of a directory and the WATFIV source code or assembled object code for the subroutines. Two different library organizations are available; one for line files and one for

sequential files. The structure of a line file library is similar to the structure of a macro library (see MTS Volume 3, System Subroutine Descriptions).

The Directory:

(1) Each entry in the directory contains the name of a subroutine in columns 1-8 and the line number of the first WATFIV statement or object record of the subroutine in columns 10-16. (Both the name and the line number must be left-justified with trailing blanks.)

(2) The line number of the first entry in the directory must be 1.

(3) The terminating entry in the directory is a string of eight (character) zeros in columns 1-8.

The Subroutines:

(1) The line number of the first statement in the subroutine must be a positive integer.

(2) The first subroutine follows the last entry in the directory.

(3) Each subroutine must be followed by a line with the /TERM control command in columns 1 through 5.

The structure of a sequential file library is similar to the DIR format for object module libraries (see the *OBJUTIL description in MTS Volume 2, Public File Descriptions, or the section "The Object File Editor" in MTS Volume 5, System Services).

The Directory:

(1) The first line of the sequential file contains the directory. This consists of a 12-byte field for each entry; the first 8 bytes contain the name of the subroutine (left-justified) followed by a corresponding 4-byte pointer to the first line of the subroutine. WATFIV uses this pointer as the read pointer for the MTS POINT subroutine when reading a member of the library (see the POINT subroutine description in MTS Volume 3).

(2) There may be at most 2730 subroutine names in the directory.

The Subroutines:

(1) The first subroutine begins with the second line in the file, immediately following the directory.

(2) Each subroutine must be followed by a line with the /TERM control command beginning in column 1.

The sequential file format is more efficient than the line file format since it minimizes the number of I/O operations needed to read the file. However, the line file format allows easy editing of source members and new subroutines may be added quite easily.

The *WATLIB routine FIVPAK should be used to compress the WATFIV-coded library subroutines to minimize storage requirements and to increase efficiency. Further details concerning FIVPAK are given in the section "Source Statement Compression Routines."

Generating a Subroutine Library

The program *WATGENLIB may be used for automatically generating WATFIV source libraries in either the line file or sequential file format. The program is invoked by the $RUN command as follows:

```
$RUN *WATGENLIB [logical unit specifications] [PAR=options]
```

where the logical I/O unit specifications are as follows:

SCARDS - WATFIV-coded subroutines and/or object modules.
SPRINT - listing of the number of library members generated and the CPU time used.
SPUNCH - the generated library.
SERCOM - error diagnostic messages.

The following options may be specified in the PAR field of the $RUN command. They must be separated by blanks or a comma.

COUNT=n "n" is the maximum number of subroutine names to be included in the library directory. WATGENLIB uses this count in allocating workspace for its internal directory. If this count is too small, then execution of the program will terminate and an error message will be printed. The default is 100.

ENTRY The ENTRY option specifies that names defined as entry points (type LR or LD in object module ESD records, or the ENTRY statement in a WATFIV source subroutine) are to be included in the library directory. NOENTRY specifies that they are not to be included. The default is ENTRY.

SORT The SORT option specifies that the library directory is to be sorted into ascending alphanumeric order and that the library members will be written in the sorted order. NOSORT suppresses the sort. The default is SORT.

Each input program segment is examined to determine whether it is an object module or a WATFIV-coded subroutine or function. If the segment is an object module, it must have at least one ESD record with a defined

symbol; the last record must be an END record. If the segment is not an object module, it is assumed to be a WATFIV-coded source routine. The first line of the routine must be either a SUBROUTINE or FUNCTION statement; the last line must be an END statement. If the ENTRY option is specified (the default), each line of the routine is examined for ENTRY statements and the corresponding entry names will be included in the library directory. Any WATFIV control commands (/COMPILE, /DATA, etc.) are ignored.

The input program segments are buffered in virtual memory while WATGENLIB constructs the library directory. When all input is processed, the directory is sorted (unless NOSORT is specified) and the output library is generated with the library members written in alphabetical order on SPUNCH. The number of members written (excluding entry points) and the total CPU time required to generate the library is printed on SPRINT.

The format of the generated library depends on the type of file assigned to SPUNCH. If this is a line file, the resulting source library will be in line file format; if this is a sequential file, the library will be in sequential file format.

The following notes apply to generating libraries:

(1) The output library file should be empty before *WATGENLIB is run.

(2) All line file libraries must be copied indexed as:

```
$COPY oldlibrary newlibrary@I
```

(3) All sequential file libraries must be copied without trimming as:

```
$COPY oldlibrary@-TRIM newlibrary@-TRIM
```

(4) WATFIV-coded source routines should be compressed to the multiple statement per line format via the *WATBLIB routine FIVPAK before *WATGENLIB is run. This will improve the compile time for the routine and will significantly reduce the amount of file space required to store it (about 60% less).

360-ASSEMBLY LANGUAGE SUBPROGRAMS

This section describes the conventions for coding subprograms in 360-assembly language for use with WATFIV. The conventions are, in fact, similar to those required by the FORTRAN-G and -H compilers. An experienced assembly language programmer should have little trouble converting existing routines to run under WATFIV.

Symbolic notation is used for registers throughout the following description; Ri means general register "i" and Fj means floating-point register "j".

Subprogram Calling Sequences

Suppose a subroutine or function subprogram "rtn" is referred to by a CALL statement or function reference in a FORTRAN source statement, e.g.,

```
     CALL rtn(arg1,arg2,arg3,...,argn)

     Y=rtn(arg1,arg2,arg3,...,argn)
```

The calling sequence generated by WATFIV for either case is

```
              CNOP  2,4
              LA    R14,RETURN
              L     R15,=V(rtn)
              BALR  R1,R15
              DC    AL1(c),AL3(addr)   argument list
              DC    AL1(c),AL3(addr)
              .
              .
              .
              DC    AL1(c),AL3(addr)
     RETURN   EQU   *
```

Each "c" is a code which describes the kind of argument list entry; each "addr" is either the address of an argument or a pointer to more information about the argument. The calling routine also provides an 18-fullword, OS-type save area.

Thus, on entry to the called subprogram:

- R15 contains the address of the entry point,
- R14 contains the normal return address,
- R13 contains the address of a save area, and
- R1 contains the address of an argument list aligned on a fullword boundary.

Moreover, it is a WATFIV convention that F6 will contain zero and R12 will contain the base address of a set of routines, constants, switches, etc., internal to the compiler.

When control is returned to the WATFIV-compiled program, it expects that:

- at least registers R5-R13 have been restored,
- the result of a function reference is returned in

(a) R0, if the function is INTEGER or LOGICAL
(b) F0, if the function is REAL
(c) (F0,F2), if the function is COMPLEX

- F6 still contains zero.

The six major categories for a code byte "c" and associated meanings of "addr" are:

(1) Unchangeable Quantity - Q: c = B'0000mmmm'

An argument which should not be changed by the called subprogram is flagged with this code byte. Constants, temporaries, DO-parameters, and assigned GOTO indices are unchangeable quantities.

The low-order four bits "mmmm" give the type of the argument according to Table 1.

For types 0-7, AL3(addr) = AL3(Q); for types 8 and 9, AL3(addr) = AL3(Q), where Q is a fullword of the form

```
DC AL1(n),AL3(Q)
```

Type	Type Number	mmmm	s-Value
LOGICAL*4	0	0000	2
LOGICAL*1	1	0001	0
INTEGER*4	2	0010	2
INTEGER*2	3	0011	1
REAL*4	4	0100	2
REAL*8	5	0101	3
COMPLEX*8	6	0110	3
COMPLEX*16	7	0111	4
CHARACTER*n n=1	8	1000	0
CHARACTER*n n>1	9	1001	0

Table 1: Type-Code Bits

(2) Variables, Array Elements - V: c = 'B1000mmmm'

Here, AL3(addr) = AL3(V), and "mmmm" is as given in Table 1. If the argument is an array element, an extra word follows in the argument list as a special indicator. This has the form

```
DC X'8C',AL3(V*)
```

where V* is the so-called STAR routine for an array of which the element is a member. STAR routines are described in the section

below. For example, if V(5) is used as a subprogram argument, the corresponding argument list entry would appear as follows:

```
DC B'10000100',AL3(V1+16),X'8C',AL3(V*)
```

where V1 is used as symbol to represent V(1). (V1 is assumed to be REAL*4).

(3) Array Name - A: c = B'1kkkmmmm'

Here, "kkk" is the number of dimensions of A, "mmmm" is as given in Table 1, and

AL3(addr) = AL3(A*)

(4) Subprogram Name - R:

If R is a subroutine, c = B'01010000'; if R is a function, c = B'0110mmmm'. In both cases, AL3(addr) = AL3(R), where R is of the form

```
DC A(R).
```

(5) Statement Number - &n: c = B'00110000'

Here AL3(addr) = AL3(n), where n is a fullword containing the address of the statement numbered "n".

(6) Argument List Terminator:

This is a special entry to mark the end of the argument list; it also provides information about the nature of the called routine.

If the called routine is to be a subroutine, c = B'00010000'; if the called routine is to be a function, c = B'0010mmmm'. The accompanying adcon contains no information.

STAR Routines for Array Arguments

All references to any array, say X, in a WATFIV-compiled program are made by means of the Subscript Testing and Addressing Routine (STAR routine), for X, called X*. The STAR routines for all arrays declared in FORTRAN source programs are constructed by the compiler. Each STAR routine contains information pertinent to an array (e.g., its dimensions, starting address, total length), and each is used at execution time for indexing into the array, for checking for out-of-range subscripts, and for passing the array to subprograms.

Knowledge of the form of a STAR routine is required when an assembler subprogram must receive (or pass) an array from (or to) a FORTRAN routine. For these purposes, it is sufficient to consider a skeleton STAR routine, say X*, of the following form:

```
            DS    0F
     X*     EQU   *-4
            DC    AL1(f),AL3(1st element in array)
            DC    AL1(s),AL3(length, in bytes, of array)
```

where "f=4k-4" ("k" is the number of dimensions), and "s" is the s-value corresponding to the type of array (see Table 1).

In a STAR routine constructed by the user to pass an array to a FORTRAN subprogram, the "f" and "s" bytes may be set to zero; the prologue in the FORTRAN routine references only the two AL3 adcons when initializing the STAR routine of the dummy array to which the calling array is passed.

The form of a full STAR routine constructed by the compiler is given below for documentation purposes.

Other Conventions for Assembler Subprograms

(1) Only the first 6 characters of CSECT, ENTRY, and EXTRN names are used by the WATFIV object-module loader; names longer than 6 characters are truncated. Names of ENTRY points and CSECTs must be unique.

(2) Blank COMMON may be referred to by the usual COM assembler feature. To refer to named COMMON, the V-type address constant name V(name of COMMON) is used.

(3) Assembler subprograms may use the CXD and DXD assembler features.

(4) A logical function returns its value in the low-order byte of R0; .TRUE. is X'FF', .FALSE. is X'00'. WATFIV stores the value of a logical variable in the high-order byte.

(5) To simulate a multiple-return statement "RETURN i", the called subprogram must search the argument list for the "i" statement-number argument. The address to which control should be returned can be determined from this argument list entry.

(6) To call a FORTRAN subprogram from an assembler subprogram, the following must be done:

- simulate a WATFIV-generated call as described above. This will include a properly constructed argument list and any required skeleton STAR routines. Provide a save area address in R13.

- pass on the contents of R12 that were passed to the assembler subprogram by a high-level FORTRAN routine.

- ensure that F6 contains zero.

- upon return from the FORTRAN subprogram, restore the assembler subprogram's registers R0-R4, R12, R15, if required, from its save area. The FORTRAN subprogram's epilogue restores only R5-R11, R13, and R14. Function values are returned in R0, F0, or F0-F2 as described in the section, "Subprogram Calling Sequences."

The special precautions for registers R12 and F6 are required since WATFIV assumes that they remain constant through the execution of a program. F6 is used for converting single-precision values to double-precision; R12 is used as a base register for many internal execution-time routines contained in compiler csect STARTA. Note 5 gives an example of the above linkage. Note that the save areas are chained using the standard OS rules.

A Compiler-Generated STAR Routine

The form of a full STAR routine generated by the compiler for an array A is as follows:

```
          CNOP  2,4
          DC    CL6'A'
     A*   BAL   R15,xrtn                 See note 1 below
          DC    AL1(f),AL3(1st element in array)  See note 2 below
          DC    AL1(s),AL3(length, in bytes, of array) See note 2 below
          DC    A(n)                    See note 3 below
          DC    A(d¹)                   Only k present
          DC    A(d²)
          .
          .
          .
          DC    B'c⁰c¹c²c³c⁴c⁵c⁶c⁷',AL3(a¹)  See note 4 below
          DC    B'c⁰0000000',AL3(a²)    Only j present
          DC    B'c⁰0000000',AL3(a³)
          .
          .
          .
```

where "k" is the number of dimensions declared for A; "j" is the number of variable dimensions; "d^1,d^2,..." are the dimensions to be used for calculating the position of an element within the array; "f=4k-4"; and "s" is the s-value from Table 1.

Notes on the STAR Routine

(1) The symbol "xrtn" stands for XA1 if k=1, or XAN if k>1. XA1 and XAN are names of subscript evaluation routines internal to the compiler, and are contained in compiler csect STARTA, addressable by R12. If the array is of type CHARACTER*n with n>1, the instruction is formed with R12 in the index position instead of the base position, i.e., "xrtn" is d(R12) instead of d(,R12). This fact is used by the error editor, ERROR.

(2) If the array is a dummy in a subprogram, then, when the subprogram is called, the prologue fills in the adcon for the first element from information supplied by the corresponding actual argument. Similarly, the prologue computes and fills in the length adcon if the dummy array has any variable dimensions.

(3) This word is present only if the array is of type CHARACTER*n with n>1; in this case, f=4K. In fact, the compiler treats such arrays as if they had k+1 dimensions where the implied first dimension is n.

(4) The first word is present if the array is a dummy array in a subprogram. Bits c^1 to c^7 indicate which, if any, dimensions are variable; c^1 is 1 if the last dimension is variable; c^2 is 1 if the second last dimension is variable, and so on. If none of c^1 to c^7 are 1, then both c^0 and a^1 are zero; otherwise, c^0 and a^1 specify the location of the last dimension which is variable, as follows:

if c^0=0, the AL3(a^1) = AL3(variable dimension)
if c^0=1, then AL3(a^1) = AL3(v), where v is defined by

```
DC A(variable dimension)
```

Bit c^0 will be 1 if the variable dimension is in COMMON or is a call-by-location subprogram parameter. The second, third, etc., words are present if there are two, three, etc., variable dimensions; the second word locates the second-to-last dimension which is variable, the third word locates the third-to-last dimension which is variable, and so on. For these words, c^0 and a^2, a^3, ..., are interpreted as above.

The set of words described in note 3 are used at execution time by the subprogram prologue to fill in the corresponding values

in the list of dimensions, and to compute the total length of the array. The prologue fills in STAR routines for dummy arrays only after it has passed down all other variables.

For example, for source statements

```
SUBROUTINE RTN(ALPHA,X,/M/,N)
COMMON K
DIMENSION ALPHA(10,K,N,5,M,12)
```

The compiler constructs the following STAR routine for ALPHA.

```
          CNOP  2,4
          DC    CL6,'ALPHA'
ALPHA*    BAL   R15,XAN
          DC    AL1(20),AL3(*-*)  1st element; filled by prol.
          DC    AL1(2),AL3(*-*)   length; filled by prologue
          DC    A(10)
          DC    A(*-*)            filled by prologue from K
          DC    A(*-*)            filled by prologue from N
          DC    A(5)
          DC    A(*-*)            filled by prologue from M
          DC    A(12)
          DC    B'10101100',AL3(M)
          DC    B'00000000',AL3(N)
          DC    B'10000000',AL3(K)
```

The compiler also constructs M and K as

```
K         DC    A(K)
M         DC    A(*-*)            filled by prologue
```

A STAR routine is stored in the local data area of the program segment in which it is declared; storage for an array which is not a dummy is allocated in the array area.

(5) Shown below is a sample WATFIV program which demonstrates some conventions used when coding subprograms in 360-assembly language for use with WATFIV.

The main program calls an assembly language subprogram RTN. RTN in turn calls the WATFIV-written subprogram NEXT. RTN passes a value for XX of 12.25 to be used in NEXT. NEXT initializes the array B and the variable Y and returns. RTN then passes a value for X back to the main program.

```
COMMON I
DIMENSION A(10)
CALL RTN(A,X)
PRINT, A,X,I
STOP
END
```

```
              SUBROUTINE NEXT(B,Y)
              COMMON J
              DIMENSION B(5)
              PRINT, 'HELLO FROM NEXT',' Y=',Y
              DO 1 J=1,5
            1 B(J)=J
              Y=-17.5
              PRINT, 'GOOD-BYE FROM NEXT',' Y=',Y
              RETURN
              END

              START
              ENTRY RTN
              USING RTN,15
RTN           STM   14,12,12(13)    Save caller's registers
              LR    2,13            Address of caller's save area
              LA    13,SAVE         New save area
              ST    13,8(2)         Link the two save areas
              ST    2,SAVE+4
              L     2,0(1)          Address of STAR routine for A
              L     3,4(2)          Address of 1st element of A
              LA    3,0(3)          Get rid of code byte
              ST    3,ASTAR+4       Save in dummy STAR routine
              L     3,8(2)          Length of array A
              ST    3,ASTAR+8       Save in dummy STAR routine
              L     2,4(1)          Address of second argument X
              ST    2,XADDR         Save it for later
              LA    1,ARGLIST       Address of argument list for call
              L     15,=V(NEXT)     To FORTRAN routine NEXT
              BALR  14,15           And away we go......
              DROP  15
              USING *,14
              L     2,XADDR         Addr of X passed from main program
              MVC   0(4,2),XX       Return value for X in main program
              L     13,SAVE+4       Old save area pointer
              LM    14,12,12(13)    Restore main program's registers
              BR    14              Return to main program
XX            DC    E'12.25'        Changed by Y=-17.5 in NEXT
*
*             Argument list for the call to NEXT, i.e., CALL NEXT(A,XX)
*
ARGLIST       DC    B'10010100',AL3(ASTAR)   Pointer to STAR routine A
              DC    B'10000100',AL3(XX)      Pointer to XX
              DC    B'00000000',AL3(0)       End of list indicator
XADDR         DC    A(*-*)          Save address of X here
SAVE          DS    18F             Save area
ASTAR         EQU   *-4             This is a STAR routine to
              DC    2A(*-*)         pass A to routine NEXT
              END
```

OBJECT MODULES FROM OTHER COMPILERS

Object modules which do not follow the WATFIV calling conventions (e.g., those produced by the FORTRAN-G compiler) may be used with WATFIV if they are called via the WATFIV object module interface WATSUB, which is available in *WATLIB. This subprogram is written in 360-assembly language and is callable only from a WATFIV program. The interface is used as follows:

(1) If the object module is a subroutine named "sub", it is called from a WATFIV program as:

```
EXTERNAL sub
CALL WATSUB(sub,n1,n2,...)
```

where "n1,n2,..." are the actual arguments for "sub".

(2) If the object module is a real function named "fun", it is called as:

```
EXTERNAL fun
x=RFUNC(fun,n1,n2,...)
```

where "n1,n2,..." are the actual arguments for "fun" and "x" is the real value returned.

(3) If the object module is an integer function named "ifun", it is called as:

```
EXTERNAL ifun
i=IFUNC(ifun,n1,n2,...)
```

where "n1,n2,..." are the actual arguments for "ifun" and "i" is the integer value returned.

The subprograms called through WATSUB, RFUNC, or IFUNC must not perform any input/output operations. Subroutines compiled with the FORTRAN-G or FORTRAN-H compilers may be used as long as none of the actual arguments are of type LOGICAL. The example below illustrates a WATFIV job using WATSUB.

```
$RUN *WATFIV
/COMPILE
      EXTERNAL SUB
         .
         .
      CALL WATSUB(SUB,A,B)
         .
         .
$CONTINUE WITH SUB.OBJ RETURN
/DATA
    .
```

```
      .
/STOP
```

where SUB.OBJ is a file containing the object module for SUB.

SOURCE STATEMENT COMPRESSION SUBROUTINES

The FIVPAK subroutine compresses "one-statement-per-line" FORTRAN source programs into "multistatements-per-line" programs usable in WATFIV. The UNPACK subroutine reverses the process. FIVPAK and UNPACK reside in the WATFIV source library, *WATLIB.

This form of source input is efficient if programs are to be stored in source form in files on disk, since programs in this form compile faster and require less disk space (approximately 60 percent less).

FIVPAK removes all blanks (except those embedded between primes) from the FORTRAN source statements. FOR example,

```
      DATA A,B/2H *,' *'/
      X=5.0
C     THIS IS A COMMENT
36    GO TO (3,8),I
```

is compressed into

```
      DATAA,B/2H*,' *'/;X=5.0
C     THIS IS A COMMENT
36    GOTO(3,8),I
```

The lines produced are sequence-numbered in increments of 10.

The following example illustrates how to call these routines:

```
CALL FIVPAK(nread,nwrite)
CALL UNPACK(nread,nwrite)
```

where "nread" is the unit number for input FORTRAN source, and "nwrite" is the unit number of output. Both routines also produce a listing of the output on SPRINT.

Both programs must be called from a program run under WATFIV, since they use CHARACTER variables. The job parameters NOEXT and NOWARN should be given on the /COMPILE control command to suppress the numerous extension and warning messages that result when WATFIV compiles either program. For example, to read lines from *SOURCE* and to punch a new deck:

```
$RUN *WATFIV 7=*PUNCH*
/COMPILE NOEXT,NOWARN
```

```
      CALL FIVPAK(5,7); STOP; END
/DATA
      one-statement-per-line program to be compressed
```

Several programs can be compressed using FIVPAK by placing a line with an asterisk "*" in column 1 between each complete program. UNPACK does not require such a "separator" line.

INTERRUPTS

This section provides information on the treatment of interrupts that may occur during the execution of a WATFIV program.

Subroutine TRAPS

Normally, WATFIV terminates execution of the program at the first occurrence of an exponent-overflow, exponent-underflow, fixed-divide, or floating-divide interrupt. However, the library subroutine TRAPS is provided to allow the programmer to accept interrupts of the above-mentioned types. Thus, with the appropriate use of the subroutines DVCHK and OVERFL, a programmer may provide, to some extent, his own treatment of interrupts.

The calling sequence is

```
CALL TRAPS(fovr,eovr,eund,fdiv,fldiv)
```

where the parameters are integer-valued arguments corresponding to the number of fixed-overflows, exponent-overflows, exponent-underflows, fixed-divide, and floating-divide interrupts the programmer wishes to trap. The arguments of TRAPS set up internal counters used by the compiler interrupt routine. This routine decrements the appropriate counter by 1 when an interrupt occurs; when any counter reaches zero, the program is terminated.

TRAPS may be called (and subsequently recalled) at any point in the main program or a subprogram to set (or reset) the interrupt counters. Arguments of TRAPS are screened so that the absolute value of any negative argument is used as a positive count, and a zero value is taken to mean that the current value of the corresponding interrupt counter should be left unchanged. If the argument is omitted, execution is terminated on the first interrupt for that type.

Examples:

(1) CALL TRAPS(0,5,7,-3,1)

sets the interrupt counters so that execution will be terminated on the occurrence of the first of the:

1st fixed overflow,
5th exponent overflow,
7th exponent underflow,
3rd fixed divide, or
1st floating divide exception following the execution of this call to TRAPS.

The statement CALL TRAPS(0,5,7,3) has the same effect.

(2)
```
LUNFLO = 100
LOVFLO = LUNFLO
CALL TRAPS(0,LUNFLO,LOVFLO)
```

sets the counts to terminate execution of the program on the occurrence of the first of the:

1st fixed overflow,
100th exponent overflow,
100th exponent underflow,
1st fixed divide, or
1st floating divide exception following the execution of this call.

(3) An execution-time statement-tracing feature or "ISN trace" is available. Internal statement number, or ISN, refers to the number of the execution statement given by the WATFIV compiler. Executable statements are numbered sequentially from the beginning of the program or subroutine, this number appears in the statement number column to the left of the source statements in a WATFIV listing. The trace is turned on using a /ISNON command and is turned off using a /ISNOFF command. Several pairs of /ISNON, /ISNOFF commands may appear in the same program.

An extremely simple example of this feature and the resultant output follows:

```
/COMPILE
ENTER STATEMENTS
      X=1.
/ISNON
        /ISNON
      X=X+1.
/ISNOFF
        /ISNOFF
      X=X+1.
/ISNON
```

```
        /ISNON
      X=X+1.
/ISNOFF
        /ISNOFF
      STOP
      END
/DATA

EXECUTION BEGINS...
*** ISN =     2  IN ROUTINE M/PROG ***
*** ISN =     4  IN ROUTINE M/PROG ***
```

This feature is very useful for tracing statement flow during the debugging process, and can be used to detect infinite loops, strange branches, etc.

Subroutines DVCHK and OVERFL

These subroutines function as follows:

```
CALL DVCHK(j)
```

where "j" is an integer variable that is set to 1 if the (pseudo-) divide-check indicator was on, or to 2 if off. After testing, the indicator is turned off. The indicator is set on when a fixed or floating-divide exception occurs.

```
CALL OVERFL(j)
```

where "j" is an integer variable that is set to reflect the most recent setting of a (pseudo-) overflow indicator. The variable "j" is set to 1 if an exponent overflow was the last to occur, to 2 if no exponent overflow or underflow condition exists, or to 3 if an exponent underflow was the last to occur. After testing, the indicator is set to 2 for no condition.

Notes

(1) The compiler interrupt routine loads the affected machine floating-point register with zero or the properly signed, largest floating-point number for exponent underflow or overflow, respectively.

(2) The five interrupt counters are initialized by the compiler to 1 at the start of each program. The divide-check and overflow indicators are not initialized; it is the programmer's responsibility to do this, e.g., by dummy calls.

(3) The terminating message is the only indication given by the compiler that interrupts have occurred. It is the programmer's responsibility to monitor these using OVERFL and DVCHK.

(4) WATFIV operates with the fixed overflow and significance interrupts masked off entirely.

(5) WATFIV automatically corrects for boundary alignment errors at execution time, but not without some resulting overhead. Thus, programmers are advised to ensure that operands are aligned properly, where possible, by steps taken at the source program level.

October 1983

INTERACTIVE FORTRAN

This section is divided into three parts: an introduction to Interactive FORTRAN (IF), followed by a detailed description of the facilities it provides, and finally several appendices giving command descriptions and examples.

The detailed description is intended to be read in a serial fashion. Information presented in later subsections tends to rely heavily on concepts presented in earlier subsections. Usually, one main concept is presented per subsection.

The appendices contain more reference material than conceptual material. The first appendix contains detailed descriptions of all of the IF commands. The second appendix contains a description of the features of the FORTRAN language supported by the IF system. The third appendix contains detailed examples of complete IF runs.

Throughout this section, reference is made to the term "the IF system." This term is meant to refer to the IF compiler, the IF editor, the IF debugging facilities, and any other facilities provided in the Interactive FORTRAN package.

INTRODUCTION

The IF compiler is a processor oriented toward interactive program development. IF enables the user to enter entire programs from a terminal or a file, to dynamically debug and correct the errors, and to save the debugged source program.

IF is an interpretive processor; it will not produce object modules and does not execute the compiled program efficiently. However, it is very flexible and useful for error-checking purposes. In many cases, bugs (program errors) which would take longer to find using the facilities provided by the other available FORTRAN processors will take only minutes to find using IF.

Compatibility with FORTRAN 66 and FORTRAN 77

There are two versions of the IF system. The *IF66 processor accepts the FORTRAN 66 standard (FORTRAN-IV) language that is supported by the FORTRAN-G and FORTRAN-H compilers (except for the extensions and

| restrictions listed in Appendix B). The *IF77 processor accepts the FORTRAN 77 standard that is supported by the VS FORTRAN compiler. Additionally, both versions of IF provide complete access to programs in the public library (*LIBRARY), and uses the standard system versions of the FORTRAN I/O and elementary function libraries. For these reasons, the transition from IF to FORTRAN-G, FORTRAN-H, or VS FORTRAN is completely straightforward. In fact, almost all programs which can be compiled and run using the IF system, can be compiled and run using either FORTRAN-G, FORTRAN-H, or VS FORTRAN without changes (and vice versa).
|
| It must be emphasized that the IF system is not intended to replace the existing processors, only to complement them. Once a program has been thoroughly debugged using the IF system, then it should be compiled using an object code producing compiler (such as FORTRAN-G, FORTRAN-H, or VS FORTRAN) in order to benefit from the superior execution speed attained with the object code produced.

The Beginning IF Programmer

The beginning IF programmer should understand that he will be able to compile and debug FORTRAN programs by learning how to use only a small subset of the features provided by the IF system. It is not necessary to have a complete knowledge of all of the features available in order to be able to use IF effectively.

As an aid to the beginning IF programmer, the remainder of this description has been divided into two parts. The first part consists of eight subsections that contain material which is basic to the understanding of the IF system. After reading these eight subsections and reviewing the examples in Appendix C, one should be able to use IF effectively.

Following these eight subsections, there is a second part consisting of eight more subsections that contain supplementary information corresponding to each of the topics covered in the first eight subsections. The supplementary material is not basic to the understanding of the IF system, but it should be read after the user has had some experience with IF.

A Few Definitions

A "routine" is one main program, or one SUBROUTINE subprogram, or one FUNCTION subprogram, or one BLOCK DATA subprogram. Interdependent routines collectively form a "program."

"FDname" refers to any MTS file or device name.

"Csect" refers to any object module such as might be produced by a FORTRAN compiler like FORTRAN-G. The terms "csect" and "external routine" are used interchangeably.

"Fixed format" refers to the standard FORTRAN statement format (statement label in columns 1-5; statement body in columns 7-72; continuation character in column 6; optional sequence field in columns 73-80).

"Free-format" FORTRAN statement format implies that statement labels are not restricted to appearing in columns 1-5, and that statement bodies are not restricted to appearing in columns 7-72, but rather they may appear in any columns, provided the statement label precedes the statement body.

IMMEDIATE EXECUTION

Invoking the IF System

To invoke the IF system, enter the MTS command "$RUN *IF66" or "$RUN *IF77". IF will respond by printing a line indicating the date of the version of the system which the user has invoked. In this example and in the other examples given is this section, user input is shown in lowercase.

```
# $run *if66
# EXECUTION BEGINS
* IF(NOV80)
*
```

Immediate Execution Mode

The initial mode of the IF system is known as "immediate execution mode" or simply "immediate execution." The user can always recognize immediate execution because the prefix character will be an asterisk (*). The prefix character is not entered by the programmer; it is generated by the IF system.

The user will notice, as he reads further, that the IF system has several identifiable modes each with a unique prefix character. He should learn to recognize the mode which IF is in by the prefix character which is presented.

Immediate execution mode is basically a "command mode." The asterisk prefix tells the user that IF is prepared for him to enter an IF command.

There are many IF commands, just as there are many MTS commands. Some IF commands even have names which are the same as MTS command names and serve similar functions. IF commands must begin with the command prefix character (/); the command prefix character cannot be omitted.

The following illustration shows how the /STOP command can be used in immediate execution to terminate the IF run:

```
* /stop
# EXECUTION TERMINATED
#
```

COMPILATION OF ROUTINES

Compiling FORTRAN programs or routines using the IF system is a process analogous to that required to compile routines using a conventional compiler (e.g., FORTRAN-G or WATFIV), provided no compilation errors are detected. During compilation a series of FORTRAN statements is read by the IF system, and code is generated which can subsequently be executed by IF (however, no object code is produced). Execution of compiled routines does not occur until the programmer issues a command specifying the routine in which execution is to begin.

Commands for Compiling: /COMPOSE and /COMPILE

Statements may be compiled from:

(1) any MTS file,
(2) any device such as a magnetic tape or a card reader, or
(3) "on-line" conversational input from a terminal.

Compilation is initiated by issuing either a /COMPILE command or a /COMPOSE command. The choice between these commands is based on the format of the FORTRAN source: /COMPILE is used when the source is in fixed format, and /COMPOSE when the source is in free-format. The result in either case is a program which is prepared for execution.

Creating FORTRAN Programs: The /COMPOSE Command

The /COMPOSE command is found to be most useful when in the process of creating a FORTRAN program (i.e., the program may be written on coding sheets, for example, but does not yet exist in machine-readable form). A natural way to enter a program is depicted in the following illustration. Lines of the program are entered conversationally, in free-format, one at a time:

Page Revised February 1988 October 1983

```
* /compose
      1_ subroutine assign(a,b)
  ROUTINE NAME: ASSIGN
      2_ a=b
      3_ return
      4_ end
*
```

The /COMPOSE command causes a number prefix to be printed as a prompt for each statement (like the $NUMBER command in MTS). Entry of an END statement causes the number prefixing and compilation to cease. The line which reads "ROUTINE NAME: ASSIGN" was printed by the IF system.

Compiling Existing FORTRAN Programs: The /COMPILE Command

The /COMPILE command is used for compiling programs which are already in machine-readable form (e.g., in a file, on cards, or on a tape). Programs to be compiled with the /COMPILE command must consist of fixed format statements (i.e., label in columns 1-5, statement body in columns 7-72, continuation character in column 6).

Compiling from an MTS Line File

To compile fixed format FORTRAN routines from an MTS line file simply specify the name of the line file on the /COMPILE command (e.g., "/COMPILE MYFILE"). The user must have both read and write permit access to the file. Statements will be compiled from the specified file until an end-of-file is encountered (therefore, more than one routine can be compiled using a single /COMPILE command).

```
* /compile draw.f
  ROUTINE NAME: DRAW
  ROUTINE NAME: PUT
*
```

In the above illustration, two routines (DRAW and PUT) are compiled from the MTS line file named DRAW.F.

Compiling from Any File or Device

To compile fixed format routines from any MTS file or device (not necessarily a line file), simply include the keyword FROM immediately before the file or device name as in the following example:

```
* /compile from seq
   ROUTINE NAME: MAIN
   ROUTINE NAME: PURGE
   ROUTINE NAME: ADD
   ROUTINE NAME: SEARCH
*
```

In the above example, four routines were compiled from the sequential file named SEQ.

Compilation Errors and Editing

Up to now no consideration has been given to what happens when a compilation error occurs. For compilation errors, the action taken by IF differs markedly from the action taken by conventional compilers.

When a compilation error is encountered, IF immediately invokes the MTS editor upon the routine containing the error.

Using the editor command language, the programmer can easily correct the statements of the routine which are in error (as if he were editing a file using the system editor). Then, by entering an editor RETURN command, control will be returned from the editor to the IF system so that recompilation of the routine incorporating any modifications made using the editor will automatically take place.

Study the following example closely:

```
* /compose
      1_ subroutine title
  ROUTINE NAME: TITLE
      2_ write(6,6?                    ...(1)
  WRITE(6,6?                           ...(2)
           $
  /TITLE:2/ - ERROR:  EXPECTING ")"
: change '?')'                         ...(3)
:       2.          WRITE(6,6)
: return                               ...(4)
      3_ 6 format('1*****')            ...(5)
      4_ return
      5_ end
*
```

Explanation of the above example:

(1) Programmer accidentally types a question mark instead of a right parenthesis while entering statement 2.
(2) IF responds by:
    (a) echo printing the line in error.
    (b) placing a marker ($) under the character in error.

(c) printing a diagnostic message.
(d) invoking the editor. Notice that the prefix changes to a colon (:) indicating edit mode.

(3) The programmer issues the editor CHANGE command to change the "?" To a ")".
(4) The programmer issues the editor RETURN command. This causes recompilation incorporating the change to statement 2 to take place.
(5) Recompilation was successful, and compilation resumes normally. The programmer enters lines 3, 4, and 5 of the routine.

The editing procedure is described in more detail in the next subsection.

Useful Commands Related to Compiling Routines

The /LIST Command

The /LIST command lists a routine, or a range of lines of a routine, on a specified file or device.

```
* /compile from myfile2
  ROUTINE NAME: SQROOT
* /list sqroot
*     1.          SUBROUTINE SQROOT(R)
*     2.          R=SQRT(R)
*     3.          RETURN
*     4.          END
*
```

The /LIST command is useful because statements which are compiled from files may be assigned new line numbers by IF, and these new line numbers may be viewed using the /LIST command. In general, the /COMPILE command preserves line numbers, while the /COMPOSE command does not.

The /COPY Command

The /COPY command copies a routine, or a range of lines of a routine, to a specified file or device. The statements in the routine are copied without necessarily preserving their line numbers.

```
* /compile from *tape*
  ROUTINE NAME: SQROOT
* /copy sqroot *sink*
*       SUBROUTINE SQROOT(R)
*       R=SQRT(R)
*       RETURN
*       END
*
```

Routines created by using the /COMPOSE command can be permanently saved in an MTS file with the /COPY command.

The /DISPLAY ROUTINES Command

The /DISPLAY ROUTINES command prints the name, the line range, and the type (i.e., BLOCK DATA, FUNCTION, MAIN, or SUBROUTINE) for each routine which has been compiled.

```
* /display routines
* "PUT"  RANGE=(282.,296.) TYPE=SUBROUTINE
* "DRAW"  RANGE=(1.,281.) TYPE=SUBROUTINE
* "MAIN"  RANGE=(15.,311.) TYPE=MAIN
* "XCXC"  RANGE=(1.,9.) TYPE=FUNCTION
*
```

The /DESTROY Command

Any routine which has been compiled under IF may be destroyed by specifying the routine name on the /DESTROY command. Destroying a routine never has any affect on the contents of files.

```
* /destroy fodder peat
*
```

The /DESTROY command in the above illustration destroys the routines named FODDER and PEAT.

The /SET ECHO=ON,OFF Command

By default, statements are not echo printed as they are read during compilation. To force echo printing, the user should issue the /SET ECHO=ON command prior to issuing the /COMPILE or /COMPOSE command.

The /SET LENCHK=ON,OFF Command

When compiling using the /COMPILE command, all lines longer than 72 characters are flagged by IF with warning messages. Thus, if one tries to compile a program containing sequence fields (sequence

numbers in column 73-80 of source statements), one warning message is printed for each sequence field. To suppress warning messages for lines which are greater than 72 characters long, the user should enter the /SET LENCHK=OFF command prior to issuing the /COMPILE command.

The /SET WARN=ON,OFF Command

By default, warning messages are always printed during compilation. To suppress all warning messages, the user should issue the /SET WARN=OFF command prior to issuing the /COMPILE or /COMPOSE command. The /SET WARN=OFF command has no effect on error messages.

EDITING ROUTINES

Edit Mode

The editor may be invoked either implicitly or explicitly. As illustrated in the previous subsection, the editor is invoked implicitly by the IF system itself whenever an error is detected during the compilation of a routine, but it may also be invoked explicitly by the programmer through the /EDIT command. Regardless of the reason for invoking the editor, the user will be able to identify edit mode because the prefix character will be a colon (:). Programmers not familiar with the editor and the editor command language are referred to MTS Volume 18, The MTS File Editor.

Implicit Invoking of the Editor: Compilation Errors

When a compilation error is detected, IF reacts by automatically invoking the editor upon the routine containing the error. The invoking is implicit in the sense that it happens automatically; it does not require a /EDIT command.

When the editor is invoked implicitly, the current line pointer in the editor will generally be predefined to correspond to the line number of the statement in error.

With the editor command language, modifications can be made to the routine to correct the compilation error: any statements may be changed, deleted, inserted, or replaced. Subsequently, by entering either a null line or a RETURN editor command, the editing procedure will be terminated, and recompilation of the routine incorporating any modifications will automatically take place.

```
* /compose
      1_ subroutine ploter
  ROUTINE NAME: PLOTER
      2_ common coords(2,100)
      3_ abcisa=coords(1,1
  ABCISA=COORDS(1,1
                  $
  /PLOTER:3./ - ERROR:  EXPECTING ")"
: change ',1',1)'
:       3.          ABCISA=COORDS(1,1)
: return
      4_
```

If the modifications successfully correct the compilation error, then compilation of the routine will resume normally as if an error had never occurred (as was the case above where the compilation resumed at line 4). But if either the modifications fail to correct the compilation error or the modifications introduce some new error into the routine, then the IF system simply reinvokes the editor (after producing a suitable error message).

Explicit Invoking of the Editor: The /EDIT Command

The editor may be invoked explicitly upon any routine by specifying the name of the routine on the /EDIT command (much like editing a file in MTS command mode).

When the editor is invoked explicitly, the current line pointer in the editor will be predefined to correspond to the line number of the first statement in the routine being edited.

Using the editor command language, any modifications can be made to the logic of the routine: statements may be changed, deleted, inserted, or replaced. Subsequently, by entering either a null line or a RETURN editor command, recompilation of the routine incorporating any new modifications will take place.

```
* /edit gander
: line 11
:      11.          REAL RUNWAY(10,10)
: change '10,10'20,20'
:      11.          REAL RUNWAY(20,20)
: (null line entered)
*
```

If a routine is being created using the /COMPOSE command, then - as shown in the following example - it is possible to edit the routine while compilation is in progress.

```
* /compose
      1_ namelist folks/dad,mim/
  ROUTINE NAME: MAIN
      2_ /edit
: change 1 'mim'mom'
:       1.          NAMELIST FOLKS/DAD,MOM/
: return
      2_
```

Free-Format Entry of FORTRAN Statements in the Editor

Once a routine has been compiled, it is at all times maintained in fixed format, regardless of the format in which it was originally entered (statements entered for compilation in free-format using the /COMPOSE command are automatically converted to fixed format by IF). This means that whenever the user edits a routine, he is editing a standard fixed format version of the routine. When invoked by IF, the editor makes life easier by allowing him to enter FORTRAN statements in either fixed format or free-format -- FORTRAN statements entered in free-format are automatically converted to fixed format lines.

```
: print 95 96
:      95.    88888 IF(NUM.EQ.10)GOTO 99
:      96.          TOTAL=TOTAL+A(NUM)
: insert 95
? 50 continue
? (end of file)
: print 95 96
:      95.    88888 IF(NUM.EQ.10)GOTO 99
:      95.25     50 CONTINUE
:      96.          TOTAL=TOTAL+A(NUM)
:
```

In the illustration above, the statement which reads "50 CONTINUE" was entered in free-format (notice that the statement body starts in column 4) and was automatically converted to fixed format by the editor (statement body starting in column 7).

Bypassing Recompilation: The IF Command

The editor command IF causes the editor to return control directly to the IF system; the automatic recompilation which normally happens when control is returned from the editor to IF is bypassed. Subsequently, the IF system will be in either immediate execution mode or suspended execution mode (depending on which mode the editor was invoked from).

The IF command is actually an "escape mechanism" to prevent "locking in" to the editor. For instance, if a programmer does not understand the cause of a compilation error then it is unlikely that he can correct it; a programmer in this situation can use the IF command to escape from the editor (entering a null line or a STOP command would result in the editor being reinvoked for the same error). This situation is shown in the following example:

```
* /compose
      1_ i=1
  ROUTINE NAME: MAIN
      2_ print,,i
  PRINT,,I
        $
  /MAIN:2/ - ERROR:  EXPECTING EXPRESSION
: (null line entered)
  PRINT,,I
        $
  /MAIN:2./ - ERROR:  EXPECTING EXPRESSION
: if
*
```

It is not possible to execute a routine which contains a compilation error (editing must be explicitly completed using the /EDIT command in this case).

EXECUTION OF ROUTINES

Executing Routines

Any routine which has been prepared for execution using either the /COMPOSE or the /COMPILE command is eligible for execution. As might be anticipated, more freedom and flexibility is provided when running routines under the control of the IF system than when running programs in the conventional manner. It is possible, for example, to begin execution of a routine which is itself incomplete (missing statements), in order to test that section of the routine which has been coded.

Once the programmer has begun execution of a routine, the IF system allows the execution flow to continue until one of the following occurs:

(1) normal program termination occurs (e.g., STOP statement executed),
(2) the attention key is pressed, or
(3) some condition is encountered which the IF system cannot resolve and which therefore blocks the normal flow of execution.

In each of these cases, the execution flow is said to "suspend" while the IF system enters a mode which allows the user to examine and perhaps modify his routines. Some of the most common conditions which cause execution to suspend are:

(1) an attempt to call a routine which has not been defined,
(2) an attempt to perform a calculation involving an undefined variable,
(3) an attempt to divide by zero, or
(4) an attempt to reference an array element outside the bounds of the array.

Invoking Main Routines: The /RUN Command

Execution of a main routine can be initiated by issuing a /RUN command. For example:

```
* /compose
      1_ write(6,6)
  ROUTINE NAME: MAIN
      2_ 6 format('how do you like that?')
      3_ return
      4_ end
* /run main
  HOW DO YOU LIKE THAT?
*
```

To assign units on the /RUN command, one does so (in the usual MTS manner) as indicated in the example following:

```
* /run main 6=data 7=*print* 9=-plot
```

SUSPENDED EXECUTION

Suspended Execution Mode

The IF system enters suspended execution mode whenever some condition occurs during the execution of a routine which requires programmer interaction (e.g., an error encountered during execution is a condition which requires programmer interaction).

```
* /run speak
  /SPEAK:61./ - ERROR: VARIABLE "VOICE" IS UNDEFINED
+ /list 61 61
+    61.          DX(I)=D(VOICE)-D(VOICE-1)
+
```

In the above example, an attempt to use an undefined variable in a computation resulted in a suspension of execution at line 61 of routine SPEAK. In the example the programmer issued the /LIST command, specifying that the statement at which execution actually suspended (line 61) is to be printed. The user can recognize suspended execution mode because the prefix character becomes the plus sign (+).

The routine in which the suspension occurs is known as the "currently active routine." The statement at which execution actually suspends is known as the "current point of suspension." In the example above, SPEAK is the currently active routine, and line 61 is the point of suspension. Normally, when a suspension occurs, the active routine and the point of suspension are printed enclosed within slashes (as above).

Similarity Between Immediate Execution Mode and Suspended Execution Mode

Suspended execution mode (prefix +) and immediate execution mode (prefix *) are very nearly equivalent. In fact, all of the commands which can be entered in immediate execution can be entered in suspended execution.

The main difference between immediate execution and suspended execution is that immediate execution "stands alone", while suspended execution is always associated with a routine (the currently active routine). If the user is in suspended execution, there is always a currently active routine. If one is in immediate execution, there is never an active routine.

Entering Suspended Execution FORTRAN Statements

As well as being able to enter IF commands in suspended execution, the programmer may also enter free-format FORTRAN statements and have them executed immediately (as if they were commands). Being able to enter FORTRAN statements in suspended execution provides a natural way of examining (and changing) the status of a program during a suspension.

The association between suspended execution FORTRAN statements and the currently active routine is simple in concept, yet very effective: suspended execution statements are simply compiled within the environment of the currently active routine, then executed. This means that a suspended execution PRINT statement can be used to print the value of any variable or array element in the currently active routine, and that a suspended execution assignment statement can be used to change the value of any variable or array element in the currently active routine.

```
* /compose
      1_ do 100 i=1,200
  ROUTINE NAME: MAIN
      2_  100 nexp= i**i
      3_ return
      4_ end
* /run main
  /MAIN:2./ - ERROR:  FIXED OVERFLOW; "I" ** "I"
+ print, i
           10
+
```

The above example shows how a suspended execution PRINT statement may be used to interrogate the values of variables in the currently active routine.

Notice that suspended execution FORTRAN statements are entered in free-format.

Restarting from Suspended Execution: The /RESTART Command

After the user has investigated the status of his active routine using suspended execution FORTRAN statements, the next thing that he might want to do is to restart program execution from the point at which the suspension occurred. This can be accomplished through the /RESTART command.

```
* /compose
      1_ call sub
  ROUTINE NAME: MAIN
      2_ return
      3_ end
* /run main
  /MAIN:1./ - ERROR:  ROUTINE "SUB" IS UNDEFINED
+ /compose
      1_ subroutine sub
  ROUTINE NAME: SUB
      2_ print,'in sub @@@@'
      3_ return
      4_ end
+ /restart
  IN SUB @@@@
*
```

In the above example, execution suspends at line 1 of routine MAIN when an attempt is made to invoke an undefined routine named SUB. In this case, the programmer dynamically compiles the undefined routine (using /COMPOSE), and then issues the /RESTART command to restart execution of the active routine (MAIN) beginning with line 1.

The /RESTART command also provides a facility for restarting execution of the active routine at any statement (not necessarily the statement at which the suspension occurred). To restart execution at a particular statement in the active routine simply specify the line number of the statement on the /RESTART command.

```
* /compose
      1_ i=1
  ROUTINE NAME: MAIN
      2_ print,j
      3_ print,j+1
      4_ print,i
      5_ print,i+1
      6_ stop
      7_ end
* /run main
  /MAIN:2./ - ERROR:  J IS UNDEFINED
+ /restart 4
            1
            2
+ /main:1./ - /stop /
+
```

In the above illustration, execution suspends at line 2 when an attempt is made to print an undefined variable J. The programmer issues the /RESTART command to restart execution at line 4, causing the execution flow to bypass statements 2 and 3 (which both reference J). Execution suspends a second time as the STOP statement is reached.

Expression Statements: Free-Format Output

The PRINT statement which has been used in the previous examples is not a standard FORTRAN-IV statement, but an extension borrowed from the WATFIV compiler to facilitate free-format output. In suspended execution, an even simpler way to obtain free-format output is with the use of an "expression statement." Take a WATFIV PRINT statement, remove the "PRINT," and an expression statement is formed.

```
* REAL A(2)/1.0,2.0/
* A(1)
      1.000000
* A(2),A(2)
      2.000000         2.000000
* (A(I),I=1,2)
      1.000000         2.000000
```

One difference between the PRINT statement and the expression statement is that the expression statement cannot be labeled. Another difference is that the PRINT statement directs output to unit 6, while the expression statement always directs output to the terminal. Otherwise, they are equivalent.

October 1983

DEBUGGING FEATURES

Four of the more effective debugging features provided by the IF system are described in this subsection. Naturally, because they are debugging features, they are closely associated with suspended execution mode.

Breakpoints

In the previous subsection, the concept of "suspended execution" was introduced. Execution of a routine was shown to suspend whenever some "unexpected condition" (which required programmer interaction) occurred during the execution of a routine. In essence, the condition was unexpected, and the resulting suspension was therefore also unexpected.

A breakpoint, by contrast, is a predetermined statement at which the programmer has decided that a suspension should occur. The programmer can define a breakpoint at any executable statement by specifying the line number of the statement on the /BREAK command. When the execution of a routine reaches a statement at which a breakpoint has been set, a suspension immediately results. The statement associated with the breakpoint will not yet have been executed.

```
* /compose
      1_ print,'how'
  ROUTINE NAME: MAIN
      2_ print,'are'
      3_ print,'you'
      4_ print,'feeling?'
      5_ (null line entered)
* /break main(2)
* /run main
  HOW
  /MAIN:2./ - ***** BREAKPOINT
+ /restart
  ARE
  YOU
  FEELING?
+
```

In the above illustration, a suspension results as the execution flow is blocked by the breakpoint defined at line 2. The programmer restarts execution of the routine by entering the /RESTART command. Execution suspends a second time as the execution flow reaches the end of the statements in the program.

Breakpoints are useful for monitoring a routine as execution progresses. As an example, a programmer may have a problem in a routine, and it appears that the problem may be caused by bad parameters being

passed to a subroutine. To monitor this situation, the programmer can define a breakpoint at the CALL statement referencing the subroutine, and examine (using suspended execution statements) the parameters each time execution suspends at the breakpoint.

A list of all breakpoints currently set in all routines is printed by using the /DISPLAY BREAKPOINTS command. To remove breakpoints, use the /REMOVE command.

```
* /display breakpoints
* BREAKPOINTS IN ROUTINE MAIN
*  2.
* /remove main(2)
* /display breakpoints
* NO BREAKPOINTS ARE DEFINED
*
```

Atpoints

Atpoints are a special kind of breakpoint. When debugging a routine, often execution will suspend at the same breakpoint several times (this is particularly true when the breakpoint is embedded in a loop). In many cases, the suspended execution operations carried out at the breakpoint (such as displaying the value of a certain variable, for instance) will be the same on each pass through the breakpoint. If the programmer can anticipate such circumstances, then instead of setting a breakpoint, he can set an atpoint. The advantage of setting an atpoint is that the operations which are to be carried out when the suspension occurs can be prespecified, thus avoiding the drudgery of entering the same suspended execution statements and commands over and over each time the breakpoint is encountered.

An atpoint, then, can be thought of as a prespecified series of suspended execution FORTRAN statements and commands which are automatically merged into the execution flow. The associated statement at which an atpoint has been set will be executed following execution of the corresponding atpoint statements and commands.

Atpoints are defined by using the /AT command. Defining an atpoint is a procedure very nearly identical to compiling a routine using the /COMPOSE command. Like the /COMPOSE command, the /AT command causes automatic numbering of input lines. The line number which will be assigned to the next line in the atpoint is printed as a prompt at the front of each input line.

```
* /compose
      1_ do 10 i=1,3
  ROUTINE NAME: MAIN
      2_ 10 continue
      3_ (null line entered)
```

```
* /at main(2)
      1_ print,'i=',i
      2_ (null line entered)
* /run main
  /MAIN:2./ - ***** AT-POINT
  I=            1
  /MAIN:2./ - ***** AT-POINT
  I=            2
  /MAIN:2./ - ***** AT-POINT
  I=            3
+
```

The definition of an atpoint is terminated by a null line, or an end-of-file, or an END statement. All suspended execution FORTRAN statements and IF commands are valid within atpoints.

Atpoints, of course, do not become a permanent part of a compiled routine since they consist only of suspended execution lines and are treated as such during execution.

Normally, after the series of suspended execution statements which comprise an atpoint have been executed, execution of the routine continues automatically (resuming with the statement in the program with which the atpoint is associated). However, a suspension will occur if an error is detected during the processing of an atpoint, or if the attention key is pressed during the execution of an atpoint. Furthermore, the programmer can deliberately force a suspension in an atpoint by simply including a PAUSE statement within the atpoint definition. If a suspension occurs for any of the above reasons, the point of suspension will be the statement in the program with which the atpoint is associated.

A list of all atpoints currently defined in all routines is printed using the /DISPLAY ATPOINTS command. To remove atpoints, use the /REMOVE command.

The /STEP Command

The /STEP command causes a specified number of FORTRAN statements to be executed, after which execution again suspends.

```
* /compose
      1_ print,1
  ROUTINE NAME: MAIN
      2_ print,2
      3_ print,3
      4_ print,4
      5_ print,5
      6_ return
      7_ end
```

```
* /break main:1
* /run main
  /MAIN:1./ - ***** BREAKPOINT
+ /step
              1
  /MAIN:2./
+ /step 2
              2
              3
  /MAIN:4./
+
```

In the above example, execution suspends at line 1 as the breakpoint is encountered. The first /STEP command has no operand, which causes one FORTRAN statement to be executed. The second /STEP command causes two FORTRAN statements to be executed. See the /STEP command description in Appendix A for more details.

Qualified Variables

Qualified variables provide a simple technique by which the programmer can interrogate (and modify) the values of the variables and array elements in a routine which is not currently active. A qualified variable is formed by concatenating a routine name to a variable name, as in the following example:

```
+ print,darwin:thrush
      123.0000
+ darwin:thrush=124.0
+ darwin:thrush
      124.0000
+
```

In the example, the programmer interrogates and assigns a value to variable THRUSH of the routine named DARWIN (which is not the active routine). Variables in COMMON or which are EQUIVALENCEd may not be qualified. The user must /GET the routine containing the COMMON or the EQUIVALENCE and then /RELEASE it after accessing the variable.

EXTERNAL ROUTINES

One of the major reasons for the effectiveness of the IF system as a debugging aid is that it is possible under the control of IF to have a routine which has been compiled by IF (using the /COMPILE command, for example) invoke a routine which has been compiled by a conventional compiler (such as FORTRAN-G or FORTRAN-H). Conversely, it is possible to have a routine which has been compiled by a conventional compiler invoke a routine which has been compiled by IF.

Explicitly Loading External Routines: The /LOAD Command

The programmer may explicitly load external routines (or csects, as they are sometimes called) from any file or device by specifying the file or device name on the /LOAD command. Any object module which conforms to standard FORTRAN linkage conventions may be loaded.

```
      # $run *ftn scards=*source*          ...(1)
      # EXECUTION BEGINS
      ? real array(2)/10.0,20.0/
      ? write(6,6)(array(i),i=1,2)
      ? 6 format(1x,2g15.7)
      ? return
      ? end
      ? (end of file)
        NO ERRORS IN MAIN
      # EXECUTION TERMINATED
|     # $run *if66                         ...(2)
      # EXECUTION BEGINS
|     * IF(NOV80)
      * /load -load                        ...(3)
      * /run main                          ...(4)
           10.00000       20.00000
      *
```

Explanation of the above example:

(1) The programmer compiles a simple program using the *FTN compiler. Because he did not specify SPUNCH, the object module will be placed in the scratch file named -LOAD by default.

(2) Next he invokes the IF system and...

(3) ...explicitly loads the object module which was produced by *FTN. This is done by specifying the name of the object module file on the /LOAD command.

(4) Once the object module has been loaded, it is "defined" within IF. This means it can be invoked in the same way that any routine compiled by IF would be invoked. In the above example, the programmer invoked MAIN by using the /RUN command.

Library Searches: The /LIBRARY Command

If an "undefined routine" (i.e., a routine which has been neither compiled by IF nor loaded) is encountered during execution, then the IF system will dynamically search a library or libraries for the routine; if the routine is found in a library, then it is automatically loaded and invoked. By default, IF will search only through the public library (*LIBRARY) when such a condition occurs. However, by using the /LIBRARY command, the programmer can predefine the libraries to be searched when references to undefined routines are encountered.

```
* /library numlib
*
```

The above /LIBRARY command, for example, instructs the IF system to automatically search the libraries NUMLIB and *LIBRARY (in that order) whenever an undefined routine is referenced during execution. For more details on the /LIBRARY command refer to Appendix A.

The /UNLOAD Command

Any external routine which has been loaded by using the /LOAD command, or as the result of a library search, can be unloaded by specifying the routine name (csect name) on the /UNLOAD command.

```
* /unload main
   "MAIN" UNLOADED
*
```

For more details on the use of the /UNLOAD command please refer to Appendix A.

The /DISPLAY EXTERNAL Command

The /DISPLAY EXTERNAL command produces a list of all currently loaded external routines.

```
* /display external
* EXTERNALLY LOADED CSECTS:
* TIME ADDRESS=50D450 LENGTH=D0
* CON ADDRESS=5110C8 LENGTH=630
* MAIN ADDRESS=517450 LENGTH=8BA0
*
```

The base address of each csect, and the length of each csect are also printed (base 16).

The Advantages and Disadvantages of External Routines

Using the facilities described above, the programmer is able to integrate object routines (which have been debugged) with routines compiled by IF (which presumably have not been completely debugged). The major advantage in choosing to load external routines is that

external routines execute much more efficiently than routines which have been compiled by IF. Routines which have been compiled by IF execute interpretively -- a process which affords great debugging flexibility at the expense of execution efficiency. A lesser advantage of choosing to load external routines is that one saves compilation time.

As far as the disadvantages of using loaded routines, it must be emphasized that IF has no control over the execution of loaded routines (in contrast to the complete control IF has over the execution of routines compiled by IF itself). A loaded routine which is not thoroughly debugged, or which is called with incorrect parameters, may produce invalid results, or cause a program interrupt. Usually, when a program interrupt occurs in an externally loaded routine, messages like those in the latter part of the following example are printed:

```
* /load -load
* /DISPLAY EXTERNAL
* EXTERNALLY LOADED CSECTS:
* JINX ADDRESS=515F18 LENGTH=10D8
* /run jinx
  EXTERNAL INTERRUPT PSW=001D0009 A0516FDE
  IN CSECT JINX     OFFSET 0010C6
*
```

External Common Blocks

It is permissible to load external routines containing common block definitions. Common block definitions in externally loaded routines and common block definitions in routines compiled by IF are automatically associated.

External BLOCK DATA can also be loaded. An external csect which is being loaded is considered to be BLOCK DATA if it is defined (either within IF or externally) as a common block.

MISCELLANEOUS FEATURES

Error Messages

The error messages produced by the IF system are self-explanatory. There are four ascending levels of message verbosity. Level 0 is most terse, level 1 is normal, level 2 is slightly verbose, and level 3 is most verbose. By default, all messages are produced at normal verbosity.

The verbosity level at which messages are produced can be controlled by the programmer through the /SET MSGLVL=i command, where i=0,1,2, or 3. For instance, if the user is on a slow-speed terminal (like a Teletype), then likely he will favor a lower level of verbosity.

```
* read(5,5
* READ(5,5
*         $
* ERROR:  EXPECTING ")"
* /set msglvl=0
* read(5,5
* READ(5,5
*         $
* E:  ")"?
*
```

There also exists a message-repetition facility which allows the programmer to have the last error message repeated at a specified level of verbosity. Entering "? ±i" causes the last error message to be repeated at verbosity MSGLVL±i. Entering "? i" (unsigned) causes the last error message to be repeated at verbosity i, where i=0,1,2, or 3.

```
* real lydumb*3(10)
* ERROR:  ILLEGAL LENGTH MODIFIER *****
* ? 3
* ERROR:  ILLEGAL LENGTH MODIFIER.
*         FOLLOWING IS A TABLE SHOWING
*         THE LENGTHS WHICH VARIABLES
*         MAY ASSUME:
*        REAL:  4 OR 8
*     INTEGER:  2 OR 4
*     COMPLEX:  8 OR 16
*     LOGICAL:  1 OR 4
*
```

In the above illustration, the programmer did not understand the error message produced at normal verbosity (level 1), and entered the "? 3" "command" to have the error message repeated at the highest level of verbosity (level 3). The message-repetition facility is available in the editor as well.

SUPPLEMENT TO IMMEDIATE EXECUTION

This section presents material supplementary to that presented in the subsection "Immediate Execution." Because immediate execution mode (prefix *) and suspended execution mode (prefix +) are closely related, most of the material presented here applies to both modes.

Immediate Execution Mode

Immediate execution mode is the initial mode of the IF system. As the name suggests, any FORTRAN statements or IF commands are executed immediately upon entry.

```
* i=123
* print,i
          123
*
```

Free-Format Input

Notice that immediate execution lines are entered in free-format. Free-format implies that statement labels are not restricted to appearing in columns 1-5, and that statement bodies are not restricted to appearing in columns 7-72, but rather they may appear in any columns, provided the statement label precedes the statement body. In fact, more than one free-format statement may be entered per line; just separate the statements by a semicolon. Multiple statement lines are processed from left to right.

```
* do 10 i=1,2; print,i; 10 continue
             1
             2
*
```

There are two other significant differences between "free-format" input and conventional "fixed format" input. First, lines beginning with the character "C" are not interpreted as free-format comment lines. Instead, a statement is considered to be a comment when the first nonblank character is the free-format comment character (").

```
* " this is a free-format comment
*   " and so is this; " and so is this
*
```

Second, column 6 has no special meaning in free-format. In fixed format no line can be longer than 72 characters. Fixed format statements longer than 72 characters must be broken over several lines, and the second and subsequent lines are marked with a nonblank, nonzero character in column 6 to indicate that they are continuation lines.

Free-format lines, in contrast, can be up to 255 characters in length. A minus sign (-) appearing as the last nonblank character of a free-format statement signifies that the statement is to be continued on the next line. The prefix character changes to the minus sign as a prompt to enter the continuation line.

```
* print,-
- 3.141592
      3.141592
*
```

Valid Statements and Commands

All IF commands and all FORTRAN statements except BLOCK DATA, ENTRY, FUNCTION, and SUBROUTINE are valid in immediate execution. Commands are distinguished from FORTRAN statements because they begin with a special command prefix character (/).

```
* ram=sqrt(4.0)
* print,ram
           2.0
* /stop
# EXECUTION TERMINATED
#
```

Execution of a multiple statement line terminates with the first IF command; in other words, no further statements or commands in a line are executed following the execution of a command. Complete command descriptions are given in Appendix A.

Transiency of Statements

FORTRAN statements entered in immediate execution have a property which can be described as "transiency." That is, they are executed, and they may have some permanent effect on the immediate execution environment (such as assigning a value to a variable or explicitly typing a variable), but after the line is executed, they are forgotten.

The changes to the immediate execution environment vary according to the statement type. GOTO statements and IF statements generally have no permanent effect; assignment statements and READ statements give values to variables and array elements, both of which can be referenced in future immediate execution (or program) computations; type statements leave attributes (type, length, dimensions, etc.) that define the context in which subsequent immediate execution computations will take place.

```
* real number
* number = sqrt(2.0)
* write(6,66)number; 66 format(1x,f10.5)
     1.41421
* /attribute number
* NAME= NUMBER  TYPE= REAL *4
* EXPLICITLY TYPED, IMMEX
*
```

Labeled Statements

One consequence of the transient property of immediate execution statements is that it is not possible to label a statement on one immediate execution line, and subsequently refer to that label on another line. Statement labels are defined only for the duration of the immediate execution line. The one exception to this rule is that format statement labels are nontransient; that is, a format statement label remains defined from line to line.

```
* 10 format(' hi there mom and dad ')
* do 20 i=1,2
* ERROR:  DO-OBJECT LABEL 20 IS UNDEFINED
* do 20 i=1,2; 20 print 10
  HI THERE MOM AND DAD
  HI THERE MOM AND DAD
*
```

Effect of Errors

When an error is detected during either the syntax check or the execution of an immediate execution line, usually the only effect is that a diagnostic message is printed. All the programmer has to do is reenter the line correctly.

```
* dimension a(4)
* a(5) = 2**8
  error:  subscript number 1
          of "a"=+5; out of range
* A(4) = 2**8
* WRITE(6,82) A(4); 82 FORMAT('A4=',F5.1)
  a4=256.0
*
```

If an error is detected while the line is being executed, it is not clear whether or not the line makes any changes to the immediate execution environment. Any computations done up to the point at which the error occurred (such as a function reference) will already have

taken effect. Remember, computation normally proceeds from left to right.

Erasing Immediate Execution: The /ERASE Command

The /ERASE command provides a facility for removing the effects of all previously entered immediate execution statements. All variables, arrays, format statement labels, statement function definitions, declarations, and so on which existed and were defined prior to the /ERASE command do not exist after the /ERASE command is issued. The immediate execution environment is cleared to its initial state.

```
* real i
* i= 50+50
* print,i
      100.0000
* /erase
* print,i
   ERROR:  I IS UNDEFINED
* i= 50+50
* print,i
           100
*
```

Modifying the Output Produced by an Expression Statement

The output produced by an expression statement can be modified by prefixing the expression statement with either @Z or @X (to force hexadecimal output), or @A or @C (to force character output).

```
* data ilash/'one'/
* @c ilash
  ONE
* @x ilash
  D6D5C540
* integer errrtn/1073741824/
* @x errrtn
  40000000
*
```

Attention Interrupts in Immediate Execution

Pressing the attention key while in immediate execution mode terminates the execution of the current immediate execution line (if execution is in progress) and otherwise has no effect.

```
* 1 print,'hi friend'; goto 1
  HI FRIEND
  HI FRIEND
  (attention key pressed)
  ***** ATTENTION INTERRUPT
*
```

SUPPLEMENT TO COMPILATION OF ROUTINES

This section presents material supplementary to that presented in the subsection "Compilation of Routines."

More Information About the /COMPOSE Command

Two facets of the /COMPOSE command warrant more explanation.

First, the rules for entering free-format statements into routines being compiled are the same--with one exception--as the rules for entering immediate execution lines. The one exception to the rules is that expression statements are not valid while routines are being compiled. Note that any IF command entered is executed immediately.

```
* /compose
      1_ subroutine title
  ROUTINE NAME: TITLE
      2_ write(6,6)
      3_ 6 format(' sample program')
      4_ return
      5_ /run
  SAMPLE PROGRAM
      5_ end
*
```

In the above example, a routine was run while compilation was still in progress. After execution terminates, compilation resumes normally.

Second, as well as being able to compile free-format routines from the terminal, the /COMPOSE command allows one to compile free-format statements from any file or device. Simply specify the file or device name on the /COMPOSE command.

```
* /compose regression.s
  ROUTINE NAME: MAIN
  ROUTINE NAME: NONPAR
  ROUTINE NAME: PRTILM
  ROUTINE NAME: SNITCH
*
```

The above example compiles four free-format routines from the MTS file named REGRESSION.S.

More Than One Main Program

It is permissible under IF to have more than one main routine compiled. The first main routine compiled is assigned the name MAIN. The second and subsequent main routines compiled are automatically assigned the names MAIN1, MAIN2, and so on. Entry, function, and subroutine names must be unique.

Restarting an Interrupted Compilation

If a compilation has been "interrupted," then it may be restarted by using either the /COMPOSE RESTART or the /COMPILE RESTART command. Compilation can be interrupted for a number of reasons, but the most common reason is an attention interrupt.

```
* /compile from custard.s
  ROUTINE NAME: MAIN
  ROUTINE NAME: CGROUP
  ROUTINE NAME: HGROUP
  12    FORMAT(' TAN(X)=' G15.7)
                          $
  WARNING:  OTHER COMPILERS MAY REQUIRE "," HERE
  (attention key pressed)
  ATTN
* /set warn=off
* /compile restart
  ROUTINE NAME: IDREAD
  ROUTINE NAME: SORT
*
```

In the above example, the compilation was interrupted by pressing the attention key, warning messages were disabled by issuing the /SET WARN=OFF command, and then the compilation was restarted. The /COMPILE RESTART was issued (rather than the /COMPOSE RESTART command) because the compilation was initiated by the /COMPILE command.

Undefined Statement Label References

When the END statement is entered into a routine during compilation, the IF system checks at that time to see if all statement labels which were referenced in the routine have been defined. If one or more labels

are undefined, then an error message is printed and the editor is invoked so that corrections can be made.

When the compilation of a routine is terminated by some mechanism other than the END statement (e.g., a null line or an end-of-file), the IF system defers the check for undefined labels. In this case, the fact that a label is undefined will be checked for during the execution of the routine when the label is first referenced. Thus, it is possible to use the IF system to compile and debug selected portions of an incomplete routine.

```
* /compose
      1_ goto 99
  ROUTINE NAME: MAIN
      2_ 999 print,'hello'
      3_ (null line entered)
* /run main
  /MAIN:1./ - ERROR: NO STATEMENT WITH LABEL 99
+
```

The Workfile

As free-format statements are compiled (/COMPOSE) they are converted automatically to fixed format and copied to a scratch line file known as the workfile. The fixed format statements are thereafter maintained in the workfile. This means that a /LIST command will list the version of the routine in the workfile, a /EDIT command will edit the version of the routine in the workfile, and a /COPY command will copy the version of the routine in the workfile.

For fixed format compilations (/COMPILE), the situation is slightly more complex because only for the "/COMPILE FROM" command are statements copied to the workfile. The other form of the /COMPILE command (which can only be used for compiling programs from MTS line files) does not use the workfile. Rather, in the latter case, statements are maintained in the original line file. This means that subsequent /LIST, /EDIT, and /COPY commands will all cause processing of the statements in the original file.

For the above reason, sometimes it is more prudent to use the "/COMPILE FROM" form of the /COMPILE command (even if the source code is stored in a line file) for by doing so processing will affect only the duplicate copy of the program being maintained in the workfile, and the chances of damaging the contents of the original file accidentally (through editing) will be eliminated.

Saving the Source Code

After the user has created a program (/COMPOSE), or edited any compiled program, he will usually be interested in saving the debugged FORTRAN source statements permanently in an MTS file. Normally, to do this, the programmer uses the /COPY command which will copy a specified routine (or all routines) to a specified file. For example:

```
* /copy main pgmsave
```

The above /COPY command would copy the routine named MAIN to the MTS file named PGMSAVE.

Of course, if the form of the /COMPILE command which does not use the workfile was used to compile a routine, then it is not necessary to /COPY the routine. In this case, any changes made to the routine in the editor will have affected the statements in the original line file.

Tabularized Summary of Commands for Compiling

The following table may be used as a quick reminder of the forms of the /COMPOSE and the /COMPILE commands that should be used for chosen purposes.

Form of Command	Source Format	Remarks
/COMPOSE	free	Compiles free-format routines from the current input stream (the terminal by default). Statements copied to workfile.
/COMPOSE FDname	free	Compiles free-format routines from the specified file or device. Statements copied to workfile.[1]
/COMPOSE FROM FDname	free	Identical to the above form.[1]
/COMPILE filename	fixed	Compiles fixed format routines from the specified line file. "Filename" must be a line file, and one must have both read and write access to it. Statements not copied to workfile.[2]
/COMPILE FROM FDname	fixed	Compiles fixed format routines from any file or device. Statements copied to workfile.[1]

[1]Concatenation of FDnames permitted.
[2]Concatenation of FDnames permitted, but routines must not be split across concatenations.

SUPPLEMENT TO EDITING ROUTINES

This subsection presents material supplementary to that presented in the subsection "Editing Routines."

Entry of Comment Statements in the Editor

In an earlier subsection, it was stated that FORTRAN statements could be entered in either fixed or free-format in the editor; FORTRAN statements entered in free-format were automatically converted to fixed format. The same holds true for FORTRAN comments entered in the editor.

A statement entered in edit mode is considered to be a comment statement if the first nonblank character is the free-format comment character ("), or if the line begins with a "C" in column 1 followed by a nonalphanumeric, nonblank character in column 2.

```
: insert 100
? " this is a comment
? c** and so is this
? cbut this isn't
? (null line entered)
: print 100 c=3
:     100.    C     THIS IS A COMMENT
:     100.25  C** AND SO IS THIS
:     100.5         CBUT THIS ISN'T
:
```

In the above example, the first line inserted is a free-format comment, the second line inserted is a fixed format comment, and the third line inserted is not a comment (because an alphanumeric character is in column 2). Notice that the editor converts the free-format comment character (") to the fixed format comment character (C).

Multiple statement lines and free-format continuation lines are currently not supported by the IF editor. The only way to enter a FORTRAN continuation line in the editor is by entering a fixed format continuation line (i.e., a line consisting of blanks in columns 1-5 and a nonblank, nonzero character in column 6).

The SET FIXED=ON,OFF Command

By default, free-format FORTRAN lines entered in the editor are automatically converted to fixed format. This conversion can be disabled either by issuing the SET FIXED=ON editor command, or by appending the "@F" modifier to editor commands.

Editing Atpoints

It is possible to edit atpoint definitions in exactly the same way as routines are edited. To edit an atpoint simply specify, on the /EDIT command, the routine name and line number (separated by a colon) which designate the atpoint. For example:

```
* /compose
      1_ i=10
  ROUTINE NAME: MAIN
      2_ return
      3_ end
* /at main(2)
      1_ 'i=',i
      2_ end
* /run main
  /MAIN:2./ - ***** AT-POINT
  I=           10
* /edit main(2)
: replace
: 'I=',I
? 'i=',i,' miles'
:       1.    'I=',I,' MILES'
: (null line entered)
* /run main
  /MAIN:2./ - ***** AT-POINT
  I=           10 MILES
*
```

In the above illustration, the atpoint statement which reads 'I=',I is an expression statement.

SUPPLEMENT TO EXECUTION OF ROUTINES

This subsection presents material supplementary to that presented in the subsection "Execution of Routines."

Logical Unit Assignments and the /RUN Command

Each time the /RUN command is issued, all logical I/O units other than those appearing on the /RUN command automatically become unassigned. The following table lists the I/O units which can be assigned on the /RUN command, and their defaults if they are not assigned:

0-4	do not default
5	*SOURCE*
6	*SINK*
7-19	do not default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH*, if in batch mode
GUSER	*MSOURCE*
SERCOM	*MSINK*

Units are assigned on the /RUN command (in the usual manner) as follows:

```
* /run main 6=data 7=*print* 9=-plot
```

Furthermore, each time the /RUN command is issued, every routine is automatically "cleared" the first time it is invoked. Clearing a routine means setting all variables and arrays in the routine to being undefined, and then assigning initial data values to those variables and arrays which are given initial data values in DATA and explicit type statements.

Invoking Subroutines and Functions

When subroutines and functions are referenced during the execution of a program, they are automatically invoked. It is also possible to invoke subroutines and functions directly by using immediate execution (or suspended execution) FORTRAN statements.

```
* /compose
      1_ subroutine print(array,n)
  ROUTINE NAME: PRINT
      2_ dimension array(n)
      3_ write(6,6)(array(i),i=1,n)
      4_ 6 format(1x,2g15.7)
      5_ return
      6_ end
* real bb(2)
* bb(1)=99.; bb(2)=100.
* call print(bb,2)
     99.00000       100.0000
*
```

In the above illustration, an immediate execution FORTRAN CALL statement was used to invoke the subroutine named PRINT. Function subprograms can be invoked from immediate execution or suspended execution simply as function references.

```
* /compose
      1_ function add(a,b)
  ROUTINE NAME: ADD
      2_ add=a+b
      3_ return
      4_ end
* print,add(2.+3.)
  ERROR:  REFERENCING SUBPROGRAM ADD WITH TOO FEW REAL ARGUMENTS
* add(2.,3.)
      5.000000
*
```

When subroutines and functions are invoked from immediate execution mode in the manner just described, logical I/O unit assignments do not automatically become unassigned, and automatic clearing of routines does not occur. In these cases, logical I/O unit assignment and clearing may be controlled explicitly by using the /SET and /CLEAR commands described in Appendix A (if desired). Only when the /RUN command is issued do these two actions occur automatically.

SUPPLEMENT TO SUSPENDED EXECUTION

This subsection presents material supplementary to that presented in the subsection "Suspended Execution." Suspended execution (prefix +) and immediate execution (prefix *) are very similar; most of the information presented in the subsection entitled "Supplement to Immediate Execution" applies to suspended execution as well.

Valid Statements and Commands

All IF commands and all FORTRAN statements except BLOCK DATA, COMMON, ENTRY, EQUIVALENCE, FUNCTION, and SUBROUTINE are valid in suspended execution. Commands are distinguished from FORTRAN statements because they begin with a special command prefix character (/).

More Commands Related to Suspended Execution

The /CONTINUE Command

Like the /RESTART command, the /CONTINUE command will restart execution of the currently active routine. If the suspension was caused by an execution error, execution will restart at the statement following the statement at which the error occurred (i.e., at the statement following the current point of suspension). If the suspension was caused intentionally (e.g., breakpoint), then execution will restart at the current point of suspension.

```
* /compose
      1_ dimension a(3)
  ROUTINE NAME: MAIN
      2_ do 10 i=1,4
      3_ a(i)=i*i
      4_ 10 continue
      5_ write(6,6)a
      6_ 6 format(1x,3g15.7)
```

```
      7_ return
      8_ end
* /run main
  /MAIN:3./-ERROR: SUBSCRIPT NUMBER 1 OF "A"=+4; OUT OF RANGE
+ /continue
     1.000000       4.000000       9.000000
*
```

In the above example, execution suspends at line 3 of routine MAIN when an attempt is made to store a value into nonexistent array element A(4). In this case, the programmer decides to ignore the error and let execution of his routine continue at statement 4 by issuing the /CONTINUE command. One alternate solution would have been to fix the error by using the editor (/EDIT command), and then rerun the routine by using the /RUN command.

The /GET and /RELEASE Commands

The /GET command is used to get a suspension or to make a routine active. The /RELEASE command has the opposite effect of the /GET command -- it releases or deletes a suspension.

It is possible to have more than one level of suspension. Levels of suspension may be thought of as being maintained on a "push down" stack. The /GET command adds a new suspension to the top of the stack, while the /RELEASE command deletes the top level of suspension from the stack.

```
* /get main
  /MAIN:1./
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE MAIN
+ /get draw
  /DRAW:1./
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE DRAW
+  SUSPENSION AT LINE 1. OF ROUTINE MAIN
+ /release
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE MAIN
+ /release
*
```

The rationale for having more than one level of suspension is this: it is often more convenient to examine an active routine than it is to examine a routine which is not active. While it is possible to examine a routine which is not currently active by using qualified variables (e.g., MAIN:A), it is not as convenient in general as a /GET, examine, /RELEASE combination.

It is worth noting that the /GET command is not the only way to get more than one level of suspension (e.g., invoking a subroutine from

suspended execution using a CALL statement may cause a further suspension), and the /RELEASE command is not the only way of releasing suspensions (e.g., all levels of suspension are released by the /RUN command).

The /IMMEX Command

The /IMMEX command causes unconditional return from suspended execution mode to immediate execution mode. In the course of going from suspended execution mode to immediate execution mode the prefix character changes from "+" to "*", and all levels of suspension are released.

```
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE DRAW
+  SUSPENSION AT LINE 8. OF ROUTINE MAIN
+ /immex
* /display levels
* ERROR:  THERE IS CURRENTLY NO SUSPENSION
*
```

There is really no reason for returning to immediate execution mode (since almost everything one can do in immediate execution can also be done in suspended execution), except that in immediate execution there is no chance of a conflict arising between a suspended execution variable and a compiled program variable. For example, if a variable, say BB, was explicitly typed as REAL*8 in the active routine, then an attempt to explicitly type BB in suspended execution would produce an error message.

IMMEX, a Predefined Routine

At the risk of belaboring a point, let it be stated again that aside from the prefix character, the main difference between immediate execution and suspended execution is that immediate execution "stands alone," while suspended execution is always associated with an active routine. Actually that is not quite true because immediate execution happens to be associated with a routine too. However, it is a null routine with no statements but it has a name, IMMEX (the immediate execution routine).

While IMMEX cannot be edited, listed, copied, or destroyed because it has no statements, it does have some attributes possessed by all routines. In particular, it is possible to /GET IMMEX and to /RELEASE IMMEX just like one would /GET or /RELEASE any other routine. The advantage of being able to do this is that it enables the user to return from suspended execution mode to immediate execution mode without releasing or deleting the current suspension (which would be the case if one returned to immediate execution mode either by issuing the /IMMEX command or by issuing a series of /RELEASE commands).

```
* /get main
  /MAIN:1./
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE MAIN
+ /get immex
* /display levels
*  SUSPENSION IN IMMEDIATE EXECUTION
*  SUSPENSION AT LINE 1. OF ROUTINE MAIN
* /release
+ /display levels
+  SUSPENSION AT LINE 1. OF ROUTINE MAIN
+ /release
* /display levels
* ERROR:  THERE IS CURRENTLY NO SUSPENSION
*
```

Referencing Compiled Program Labels from Suspended Execution

Executable statement labels in the active routine cannot be referenced from suspended execution. For instance, if there was an assignment statement labeled 99 in the active routine, then a "GO TO 99" suspended execution statement would not restart execution in the active routine. Rather, it would cause an error message to be produced to the effect that 99 could not be used both as a compiled program label and as a suspended execution label. The correct way to restart execution at statement labeled 99 is to issue a /RESTART #99 command.

However, it is possible to reference FORMAT statement labels in the active routine from suspended execution (or in atpoints) as the following example shows:

```
* /compose
      1_ i=12
  ROUTINE NAME: MAIN
      2_ write(6,10)i
      3_ 10 format(' number=',i2)
      4_ (null line entered)
* /run main
  NUMBER=12
+ write(6,10)i+i
  NUMBER=24
+
```

Tabularized Summary of Commands Related to Suspended Execution

The following table may be used as a quick reminder of what commands should be used for what purposes in suspended execution.

Command	Description of Command
/RESTART	Restarts program execution beginning with the statement at the current point of suspension. Execution may be restarted at any other executable statement by specifying the line number of the statement on this command (e.g., "/RESTART 80" would restart execution at the statement at line number 80).
/CONTINUE	Restarts execution of the currently active routine. If the suspension was caused by an execution error, execution will restart at the next statement (i.e., at the statement following the current point of suspension). If the suspension was caused intentionally (e.g., breakpoint), then execution will restart at the current point of suspension.
/GET	Makes a specified routine active; the first executable statement in the specified routine becomes the current point of suspension.
/RELEASE	Releases the current suspension; the previously active routine once again becomes the currently active routine.
/IMMEX	Causes an unconditional return from suspended execution mode to immediate execution mode. The prefix character changes from "+" to "*", and all levels of suspension are deleted.

SUPPLEMENT TO DEBUGGING FEATURES

This subsection presents material supplementary to that presented in the subsection "Debugging Features."

SUBPROGRAM LINKAGE TRACING

The /TRACE command can be issued to have a subprogram linkage traceback produced for the current suspension.

```
+ /trace
  CALLED FROM ROUTINE "BAKER", STATEMENT 96.
  CALLED FROM ROUTINE "ABLE", STATEMENT 11.
  INITIATED FROM IMMEDIATE EXECUTION MODE
+
```

The above traceback indicates that the currently active routine was invoked from routine BAKER at line 96, which was invoked from routine ABLE at line 11., which was invoked directly from immediate execution (perhaps using a /RUN command).

Execution Flow Tracing

The programmer can have the execution flow of his routines automatically traced by issuing the /SET FLOW=ON command. With flow tracing enabled, each time a branch is made to a labeled statement, the label is printed.

```
* /set flow=on
* /compose
      1_ goto 1
  ROUTINE NAME: MAIN
      2_ 1 goto 2
      3_ 2 goto 3
      4_ 3 return
      5_ end
* /run main
  &MAIN:#1
  &:#2
  &:#3
*
```

The routine name is printed when the routine name changes. To flow trace only certain sections of routines, turn flow tracing on or off as appropriate within atpoints.

Attribute Checking

Of all the problems which can arise in a FORTRAN program, some of the most difficult to diagnose are those which occur because of the programmer's failure to type his variables and arrays properly. Typical of the problems of this nature are instances in which a library subprogram, expecting an INTEGER*2 argument, is passed an INTEGER*4 argument. Often the result of such an oversight is that a program interrupt will occur much later during the execution of the program, making the actual cause of the problem (an attribute mismatch) very hard to detect.

There are two ways to detect attribute mismatches. The first way is to actually review type statements by using the /LIST command. A second simpler way is to use the /ATTRIBUTE command, which will print the attributes (type, length, dimensions, and so on) of specified variables, arrays, and statement labels.

```
* /compose
      1_ logical*4 dice(6,6,6,6,6,6)
  ROUTINE NAME: MAIN
      2_ common dice
      3_ read(5) dice
      4_ (null line entered)
* /attribute main:dice
  NAME= DICE   TYPE= LOGICAL *4
  DIMENSION=(6,6,6,6,6,6)
  A COMMON VARIABLE, AN ARRAY,
  AN EXPLICITLY TYPED VARIABLE
*
```

Cross-Referencing

The /REFERENCE command will print a cross-reference for any variable, array, or statement label.

```
* /compile from draw.f
  ROUTINE NAME: DRAW
  ROUTINE NAME: PUT
* /reference draw:#2
* "DRAW:#2" DEFINED AT LINE 65.,
*           REFERENCED AT LINES:
*  65. 82. 123. 147. 198. 212. 226. 235.
*
```

In the above illustration, the programmer uses the /REFERENCE command to obtain a cross-reference of statement label 2 in routine DRAW. Whenever statement labels are used as command operands (in place of line numbers), they must be prefixed by a (#).

COMMON AND EQUIVALENCE MAPS

The programmer can have common and equivalence storage maps printed for the currently active routine by issuing either the /DISPLAY COMMON or the /DISPLAY EQUIVALENCE command.

```
+ /display common
+ COMMON MAP FOR ROUTINE MAIN
+ BLOCK=BABEL
+ SIZE=      74 BYTES (BASE 10)
+ NAME=DOG   , DISPLACEMENT=       0
+ NAME=BAKER , DISPLACEMENT=       4
+ NAME=FOX   , DISPLACEMENT=      44
+
```

Advanced Example of Atpoint Usage

The example which follows demonstrates how an atpoint could be used to count the number of times the execution flow of a program passes through a certain statement. In the example, a subroutine named RUNG is being called repeatedly. It is the programmer's desire to cause execution to suspend on the twentieth call to RUNG. To do this, he sets an atpoint in RUNG at statement 1 and counts the number of times RUNG is invoked:

```
* ic=0
* /list rung(1,1)
*     1.          SUBROUTINE RUNG(Y,F,Q)
* /at rung(1)
      1_ immex:ic=immex:ic+1
      2_ if(immex:ic.eq.20)pause '20th call'
      3_ (null line entered)
* /run main
  /RUNG:1./ - ***** AT-POINT
  /PAUSE '20TH CALL'/
  /RUNG:1./ - ***** AT-POINT EXECUTION TERMINATED
+
```

There are a few interesting points in the above example. First, notice the mechanism by which the programmer counts the number of times the subroutine is called. He uses an immediate execution variable (IC) which he initializes to zero with an immediate execution assignment statement prior to running his program. IC is referred to from within the atpoint as a qualified variable (i.e., IMMEX:IC). Second, examine the way in which the programmer forces execution to suspend on the twentieth call to RUNG. He uses a FORTRAN PAUSE statement which when used in any context under IF causes execution to suspend.

SUPPLEMENT TO EXTERNAL ROUTINES

This subsection presents material supplementary to that presented in the subsection "External Routines."

October 1983

Predefined Routines

The following elementary FORTRAN functions are "predefined" to the IF system:

```
ABS    AIMAG  AINT   ALGAMA ALOG   ALOG10
AMAX0  AMAX1  AMIN0  AMIN1  AMOD   ARCOS
ARSIN  ATAN   ATAN2  CABS   CCOS   CDABS
CDCOS  CDEXP  CDLOG  CDSIN  CDSQRT CEXP
CLOG   CMPLX  CONJG  COS    COSH   COTAN
CSIN   CSQRT  DABS   DAIMAG DARCOS DARSIN
DATAN  DATAN2 DBLE   DCMPLX DCOS   DCOSH
DCONJG DCOTAN DDIM   DERF   DERFC  DEXP
DFLOAT DGAMMA DIM    DIMAG  DINT   DLGAMA
DLOG   DLOG10 DMAX1  DMIN1  DMOD   DREAL
DSIGN  DSIN   DSINH  DSQRT  DTAN   DTANH
ERF    ERFC   EXP    FLOAT  GAMMA  HFIX
IABS   IDIM   IDINT  IFIX   IMAG   INT
ISIGN  MAX    MAX0   MAX1   MIN    MIN0
MIN1   MOD    REAL   SIGN   SIN    SINH
SNGL   SQRT   TAN    TANH
```

Attempting to compile a function by the same name as a predefined function is not allowed. For example:

```
* /compose
      1_ function float(intger)
  ERROR:  ENTRY POINT "FLOAT" IS PREVIOUSLY DEFINED
:
```

However, it is possible to unload even predefined functions by using the /UNLOAD command. If a /UNLOAD FLOAT command had been issued prior to issuing the /COMPOSE command in the example above, an error condition would not have resulted.

As well as being predefined, the elementary FORTRAN functions listed above are "pretyped" (e.g., the function DABS is pretyped as REAL*8).

Several resident system subroutines are also "predefined" to the IF system. These are: ERROR, EXIT, MTS, QUIT, SYSTEM, and TRACER. A reference to one of these subroutines either from within an IF compiled routine or from within an externally loaded routine results in a suspension. Execution may be continued by using the /RESTART command (in which case execution restarts at the instruction following the call to the predefined subroutine).

```
* /run main 5=bridgedata
  WARNING:  "TRACER" CALLED FROM LOADED OR LIBRARY ROUTINE
+ /restart
  SUM OF SQUARES= 0.0001
*
```

External Suspensions

If a program interrupt occurs in an externally loaded routine, or if the attention key is pressed while an externally loaded routine is executing, then an "external suspension" occurs.

```
* /compile testprog
  ROUTINE NAME: MAIN
* /load -load
* /display external
* EXTERNALLY LOADED CSECTS:
* SUBMER   ADDRESS=50CEE8 LENGTH=108
* /run main 4=data
 (attention key pressed)
  EXTERNAL ATTENTION PSW=071D0005 2050CFDE
  IN CSECT SUBMER   OFFSET 0000F6
+ /display suspension
+  EXTERNAL SUSPENSION
+  AT LINE 97.2 OF ROUTINE MAIN
+
```

In the above illustration, an external routine named SUBMER was executing when the attention key was pressed, and an external suspension resulted. The point of suspension corresponding to an external suspension is the last statement in an IF-compiled routine which was executed (usually this will be the CALL statement or function reference which invoked the external routine).

To restart execution after an external suspension the programmer has several choices:

(1) Issuing the /RESTART command (with no operand) will restart execution beginning with the next instruction in the external routine.
(2) Issuing the /CONTINUE command will restart execution beginning with the next FORTRAN statement in the active IF compiled routine (MAIN in the above example).
(3) Issuing the /REPEAT command will restart execution beginning with the current FORTRAN statement in the active IF compiled routine. This causes the external routine to be reinvoked.

SUPPLEMENT TO MISCELLANEOUS FEATURES

This subsection presents material supplementary to that presented in the subsection "Miscellaneous Features."

October 1983

Dumps

Infrequently, when one is using the IF system, a "dump" will occur. Dumps have the following format:

```
   ***DUMP*** 20167F:XLD +002058 071D0006 C020167F
  | 509058  00000000 00508224 00508218 00000016
  | 509068  0022DC50 00506FEC 00508222 00000008
  | 509078  00000000 00218CA8 002280A0 00501000
  | 509088  0022CC50 00501394 4022DAA2 00BADADD
  | PLEASE DIRECT HARD COPY OF THIS DUMP TO
  | A COMPUTING CENTER COUNSELOR.
  | THE RELIABILITY OF  THE IF SYSTEM  FOR THE
  | REMAINDER OF THIS RUN IS QUESTIONABLE;  IT
  | IS RECOMMENDED THAT YOU /COPY ANY ROUTINES
  | THAT YOU WISH TO SAVE,  AND  THEN  ISSUE A
  | /STOP COMMAND TO TERMINATE THE CURRENT RUN
    ATTN
  *
```

The numeric information which is printed out when a dump occurs is of special interest to the staff of the Computing Center.

If a dump occurs, do not be alarmed--a dump simply means that a failure has been detected internally within the IF system. In rare cases dumps are the result of a "bug" in the IF system itself, but more often they are the result of a loaded external routine overwriting IF system control information. If you suspect a bug in the IF system is the cause of a dump, please report the problem to the Computing Center so that the problem may be corrected; bring hard-copy output if possible.

Spelling Error Detection

The IF system contains a spelling error detection facility which allows it to recognize approximately 80% of all FORTRAN keywords which are misspelled.

```
     * /compose
           1_ i=1
       ROUTINE NAME: MAIN
           2_ prnt,i
       PRNT,I
       $
       /MAIN:2./ -WARNING: "PRINT" MISSPELLED
           3_ return
           4_ end
     *
```

It should be pointed out that IF produces only a warning message when it detects a misspelled FORTRAN keyword. Although the statement is accepted, IF does not correct the misspelled keyword in the source code.

IF IN BATCH MODE

Compilation in Batch Mode

In batch mode, IF behaves much like any conventional compiler. Compilation continues to completion regardless of the number of compilation errors, and all compilation errors are clearly flagged with diagnostics in the source program listing. In batch mode, a source program listing is printed by default. The editor is not invoked in batch mode when a compilation error occurs; rather, an error message is printed and compilation continues with the next statement.

In batch mode, only an end-of-file will terminate compilation. This means that more than one routine can be compiled in sequence from the same file or device.

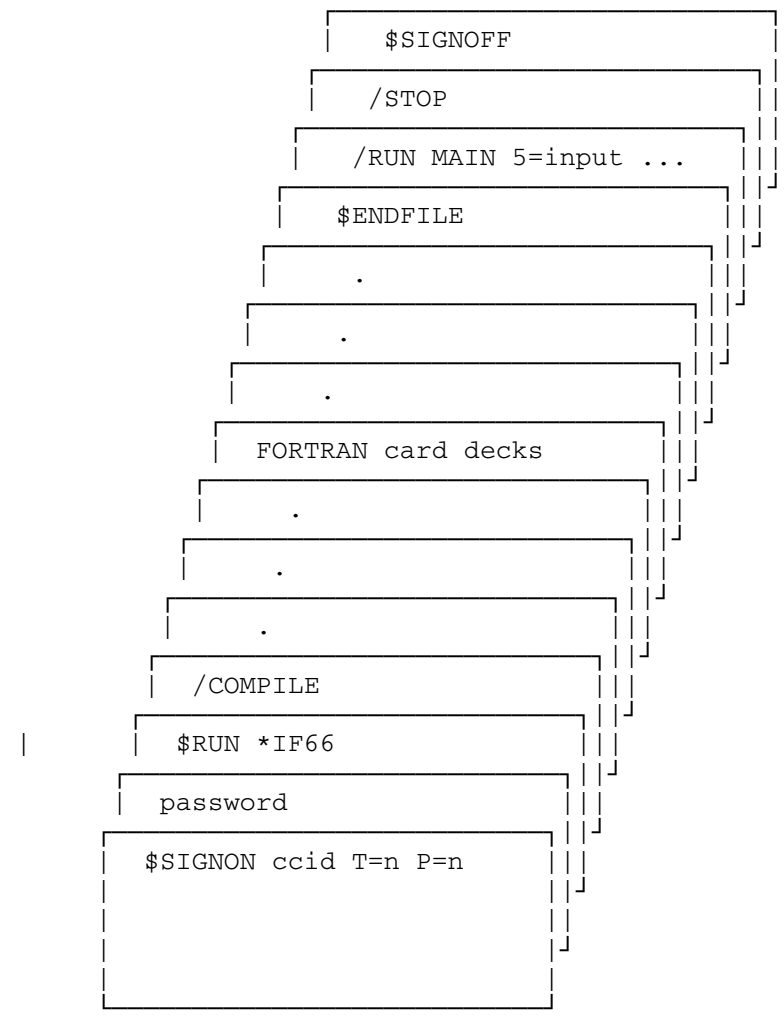
Execution in Batch Mode

The basic strategy of the IF system is to return to MTS command mode whenever an error is detected during execution. An attempt to initiate execution of a routine containing a compilation error, an attempt to use an undefined variable in a computation, and an attempt to call an undefined routine are examples of conditions which would result in a return to MTS in batch mode. Of course, an error message is produced before the return is made to MTS.

Breakpoints and atpoints can be used in batch mode just as in conversational mode. A suspension due to a breakpoint or atpoint does not cause a return to MTS command mode.

Batch Example

The following illustration shows a typical batch job setup for compiling and executing a FORTRAN source card deck using the IF system:


$SIGNOFF
/STOP
/RUN MAIN 5=input ...
$ENDFILE
.
.
.
FORTRAN card decks
.
.
.
/COMPILE
$RUN *IF66
password
$SIGNON ccid T=n P=n

APPENDIX A: COMMAND DESCRIPTIONS

This appendix contains a detailed description of each IF command. The command descriptions are presented alphabetically with each command description starting on a fresh page.

The table below may be used as a quick reminder of what commands are available. For each command, the table gives the minimum acceptable abbreviation (indicated by underlining), and a short description of the usual purpose of the command. Remember, the command prefix character (/) cannot be omitted from IF commands.

/AT	- sets atpoints
/ATTRIBUTE	- prints attributes of variables
/BREAK	- sets breakpoints
/CLEAR	- clears routines
/COMPILE	- compiles fixed format routines
/COMPOSE	- compiles free-format routines
/CONTINUE	- restarts suspended execution
/COPY	- copies routines to MTS files
/DESTROY	- destroys compiled routines
/DISPLAY	- prints miscellaneous information
/EDIT	- edits routines or atpoints
/ERASE	- clears immediate/suspended execution
/EXECUTE	- executes a section of a routine
/EXPLAIN	- explains IF commands
/GET	- makes a routine active
/HELP	- a command for beginners
/IMMEX	- returns to immediate execution
/INPUT	- reads IF commands from a file
/LIBRARY	- specifies libraries to search
/LINK	- loads and executes an object program
/LIST	- lists a routine
/LOAD	- loads external routines
/MTS	- executes an MTS command
/OUTPUT	- redefines output file or device
/REFERENCE	- cross references variables
/RELEASE	- releases current suspension
/REMOVE	- removes atpoints and breakpoints
/REPEAT	- repeats current statement
/RESTART	- restarts suspended execution
/RUN	- begins execution of main routines
/SET	- assigns I/O units; sets switches
/STEP	- steps FORTRAN statements
/STOP	- terminates the IF run
/TRACE	- prints subprogram linkage traceback
/UNLOAD	- unloads external routines
/WORKFILE	- defines workfile for IF to use

October 1983

The following standard notation conventions are used in the command prototype descriptions:

(1) Command prototype fields appearing in lowercase are generic terms which are to be replaced by an item supplied by the programmer. Command prototype fields appearing in uppercase are fields which are to be repeated verbatim in the command.
(2) An ellipsis "..." indicates that the preceding field may be repeated if necessary.
(3) Underlining indicates the minimum acceptable abbreviated form of the command or command parameter, but longer abbreviations will be accepted.

A few of the generic terms which appear within the command descriptions require explanation.

(1) "FDname" means any MTS file or device name.
(2) "linenumber" refers to the MTS line numbers associated with the FORTRAN statements in compiled routines. Each statement has a line number with which it can be referenced. If a statement is labeled, then it may also be referred to by its label; simply prefix the label with "#" (for example, "#10" refers to the statement with label "10").
(3) "rhs" means "right-hand side".
(4) "csect" refers to any object module such as might be produced by a FORTRAN compiler like FORTRAN-G. The terms "csect" and "external routine" are used interchangeably.

/AT

Purpose: (a) /AT routine(linenumber) ...
/AT linenumber ...

(b) /AT routine(#label) ...
/AT #label ...

Purpose: An atpoint is associated with each executable statement specified in the parameter list. If the routine name is omitted, the currently active routine is assumed. If the line number is omitted, the first executable statement of the specified routine is assumed.

An atpoint is a predefined series of suspended execution lines which are automatically executed by IF just prior to executing the statement with which the atpoint is associated. In conversational mode, the /AT command causes automatic numbering of input lines. The line number which will be assigned to the next line in the atpoint being defined is printed as a prompt at the front of each input line in the following form:

```
/AT MAIN(13)
     1_
     2_
```

Each line entered following the /AT command becomes part of the atpoint definition until an end-of-file, or a null line, or an END statement is entered. It is no accident that defining atpoints resembles very closely the composition of programs (see the /COMPOSE command).

Notes: Any FORTRAN statements or commands which are valid in suspended execution are valid in atpoints.

Atpoint definitions can be edited as if they were programs. The second form of the /EDIT command is used for editing atpoints (e.g., "/EDIT MAIN:44").

Atpoints may be removed using the /REMOVE command.

Atpoints may be defined only at executable FORTRAN statements. Type statements and DATA statements are not considered to be executable FORTRAN statements, although ENTRY, FUNCTION, and SUBROUTINE are.

The "/DISPLAY ATPOINTS" command prints a list of all currently defined atpoints in all routines.

Example: The following example illustrates the use of the /AT command:

```
* /compose
      1_ do 10 i=1,2
  ROUTINE NAME: MAIN
      2_ 10 continue
      3_ stop
      4_ end
* /at main(#10)
      1_ ' i=',i
      2_ end
* /run main
  /MAIN:2./ - ***** AT-POINT
   I=            1
  /MAIN:2./ - ***** AT-POINT
   I=            2
  /MAIN:3./ - /STOP /
+
```

/ATTRIBUTE

Prototype: (a) /ATTRIBUTE routine:variable ...
/ATTRIBUTE variable ...

(b) /ATTRIBUTE routine:#label ...
/ATTRIBUTE #label ...

Purpose: This command prints the attributes (type, length, dimensions, etc.) of the variable names and statement labels specified in the parameter list. If the routine name is omitted, the currently active routine is assumed.

Examples: /ATTRIBUTE CONT SUBP:II

would print the attributes of the variable CONT in the currently active routine, and the variable II in routine SUBP. The attributes are printed in the following format:

```
NAME= CONT   TYPE= INTEGER *2
DIMENSION=(10)
AN ARRAY, A VALUE ASSIGNED ITEM

NAME=    II   TYPE= INTEGER *4
A COMMON VARIABLE
```

/ATTRIBUTE #99

would print the attributes of statement label 99 in the currently active routine in the following format:

```
NAME=    99   TYPE= LABEL
A FORMAT LABEL, A COMPILED PROGRAM LABEL
```

October 1983

/BREAK

Prototype: /BREAK routine(linenumber) ...
/BREAK linenumber ...

Purpose: A breakpoint is defined at each executable statement specified in the parameter list. If the routine name is omitted, the currently active routine is assumed. If the line number is omitted, the first executable statement of the specified routine is assumed.

When a breakpoint is encountered during program execution, execution will suspend at the statement at which the breakpoint was defined. The statement at which the breakpoint was defined will not yet have been executed. The /RESTART and the /STEP commands are normal ways to restart execution after a suspension caused by a breakpoint.

Notes: Breakpoints may be removed using the /REMOVE command.

Breakpoints may only be defined at executable FORTRAN statements. See the /AT command description.

The "/DISPLAY BREAKPOINTS" command prints a list of all currently defined breakpoints in all routines.

Examples: /BREAK 23.4

would define a breakpoint at line 23.4 of the currently active routine.

/BREAK SUB1(13) SUB2(#99)

would define breakpoints at line 13.0 of routine SUB1, and at the statement labeled 99 of routine SUB2.

/CLEAR

Prototype: (a) /CLEAR routine ...

(b) /CLEAR *

Purpose: The /CLEAR command clears routines to their initial states. A cleared routine is equivalent in every sense to a newly compiled routine (except for possible atpoints and/or breakpoints which remain defined). Clearing a routine is accomplished in two stages: first, all variables and arrays in the routine are set to being undefined, and then all variables and arrays which are given initial values in either DATA statements or explicit type statements are assigned those initial values.

The first form of the /CLEAR command clears each routine specified in the parameter list. If the routine name is omitted, the currently active routine is assumed. If the parameter is "*", all routines compiled under IF are cleared.

Notes: When execution is initiated by the /RUN command, each routine which has been compiled under IF is implicitly cleared the first time it is invoked. When execution is initiated by some method other than the /RUN command, the /CLEAR command is the only method of clearing routines.

When a routine is cleared, the suspended execution data environment associated with the routine is also cleared. All suspended execution variables, arrays, format definitions, and so on become undefined. Note that the /CLEAR command has no effect on the immediate execution data environment. However, the /ERASE command can be used to clear both the immediate execution and suspended execution environments.

The /CLEAR command has no effect on externally loaded routines. To clear an externally loaded routine, first /UNLOAD it, and then /LOAD a fresh copy of it.

Example: /CLEAR SUB1 LINPG

would cause the routines named SUB1 and LINPG to be cleared.

October 1983

/COMPILE

Prototype: (a) /COMPILE

(b) /COMPILE filename

(c) /COMPILE FROM FDname

(d) /COMPILE routine FROM FDname

(e) /COMPILE RESTART

Purpose: To compile a routine or routines consisting of fixed format FORTRAN source lines. Compilation continues until an end-of-file is encountered.

Prototype (a) of this command compiles a routine from the current input file or device. The current input file or device is defined by the /INPUT command, and defaults to *SOURCE*. In conversational mode, this form of the /COMPILE command causes automatic numbering of input lines (if the current input source is the terminal). The line number which will be assigned to the next statement in the routine being compiled is printed as a prompt at the front of each input line in the following form:

```
/COMPILE
     1_
     2_
```

In conversational mode, this first form of "on-line" compiling should be avoided, simply because the /COMPOSE command is more convenient (as it accepts free-format lines). In batch mode, more than one routine may be compiled using a single /COMPILE command.

Prototype (b) of this command causes a routine or routines to be compiled from the specified file. Because the file may be used for editing purposes, it must be a line file, and the user must have both read and write permit access to it.

Prototype (c) of this command compiles a routine or routines from the specified file or device. For this form, all the user needs is read access for the specified file or device. The keyword FROM must be included.

Prototype (d) of this command causes compilation of the specified routine to continue by including fixed format FORTRAN source lines from the specified file or device

(statements are added to the end of the specified routine). This is possible only if the specified routine does not yet contain an END statement. The keyword FROM must be included, and the file or device name cannot be omitted.

Prototype (e) causes compilation to restart if compilation was interrupted, for example, by an attention interrupt. Compilation continues from the same file or device as if it had not been interrupted.

Notes: After the first statement of each routine has been compiled, IF prints the routine name in the following form:

```
ROUTINE NAME: xxxxxx
```

More than one main routine may be compiled. Main routines other than the first are automatically given names MAIN1, MAIN2, and so on. Subroutine, function, and entry point names must be unique.

To find out the names of all the routines compiled by IF, use the "/DISPLAY ROUTINES" command.

The /COPY command can be used to save the edited source version of any routine in a MTS file.

Examples: /COMPILE

in batch would compile fixed format FORTRAN programs from the card reader until a $ENDFILE was encountered.

/COMPILE FILE1

would compile routines from the MTS file named FILE1.

/COMPILE FROM *TAPE*

would compile routines from the magnetic tape *TAPE*.

October 1983

/COMPOSE

Prototype: (a) /COMPOSE

(b) /COMPOSE FDname

(c) /COMPOSE FROM FDname

(d) /COMPOSE routine FROM FDname

(e) /COMPOSE RESTART

Purpose: To compile a routine or routines consisting of free-format FORTRAN source lines. Compilation continues until an end-of-file is encountered.

Prototype (a) of this command compiles a routine from the current input file or device. The current input file or device is defined by the /INPUT command, and defaults to *SOURCE*. In conversational mode, this form of the /COMPOSE command causes automatic numbering of input lines (if the current input source is the terminal). The line number which will be assigned to the next statement in the routine being composed is printed as a prompt at the front of each input line in the following form:

```
* /COMPOSE
      1_
      2_
```

In conversational mode, this type of "on-line" composing can also be terminated by entering a null line, an END statement, or an attention interrupt. In batch mode, more than one routine may be compiled using a single /COMPOSE command.

Prototypes (b) and (c) of this command are identical. They cause a routine or routines to be compiled from the specified file or device.

Prototype (d) causes compilation of the specified routine to continue by including free-format FORTRAN source statements from the specified file or device (statements are added to the end of the specified routine). This is possible only if the specified routine does not yet contain an END statement. The keyword FROM must be included, and the file or device name cannot be omitted.

Prototype (e) causes compilation to restart if composing was interrupted, for example, by an attention interrupt.

Compilation continues from the same file or device as if it had not been interrupted.

Notes: After the first statement of each routine is compiled, IF prints the routine name in the following form:

```
ROUTINE NAME:  xxxxxx
```

More than one main routine may be compiled. Main routines other than the first are automatically given names MAIN1, MAIN2, and so on. Subroutine, function, and entry point names must be unique.

At any time, to find out the names of all the routines compiled by IF, use the "/DISPLAY ROUTINES" command. The /COPY command can be used to save the edited source version of any routine in a MTS file.

Examples: /COMPOSE

in batch would compile free-format FORTRAN source programs from the card reader until a $ENDFILE was encountered.

/COMPOSE FILE1(1,100)+FILE2

would compile the free-format FORTRAN source programs in the MTS files FILE1(1,100) and FILE2.

/COMPOSE MAIN FROM *SOURCE*

would cause compilation of the routine named MAIN to continue. In this case FORTRAN source lines would be read from *SOURCE* and would be added to the end of routine MAIN. This would not be possible if routine MAIN already contained an END statement.

October 1983

/CONTINUE

Prototype: /CONTINUE

Purpose: To continue execution of a suspended program.

If the suspension was caused by an error during program execution, execution will restart at the statement following the statement in error. (See also the descriptions of the /REPEAT and /RESTART statements.)

If the suspension was intentionally caused by either a /BREAK, /AT, /STEP, or /GET command, then execution will restart at the current statement.

Note: The current statement, that is the statement at which execution is suspended, is available using the "/DISPLAY-LEVELS" command.

Example: The following example illustrates how the /CONTINUE command can be used to restart execution of a routine after a suspension caused by a breakpoint.

```
* /compose
      1_ i=1
  ROUTINE NAME: MAIN
      2_ print,i
      3_ return
      4_ end
* /break main(2)
* /run main
  /MAIN:2./ - ***** BREAKPOINT
+ /continue
             1
*
```

/COPY

Prototype: (a) /COPY routine(linenumber1,linenumber2) FDname

(b) /COPY routine(#label1,#label2) FDname

(c) /COPY * FDname

Purpose: Prototype (a) of this command copies a routine, or a range of lines of a routine, to the specified file or device. The statements in the routine are copied without necessarily preserving their line numbers. This command is very similar to the MTS $COPY command. If the routine name is omitted, the currently active routine is assumed. If "linenumber1" is omitted, the first statement in the routine is assumed. If "linenumber2" is omitted, the last statement in the routine is assumed. If "FDname" is omitted, the current output file or device is assumed. The current output file or device, which can be redefined by the /OUTPUT command, defaults to *MSINK*.

Prototype (b) is similar to prototype (a) except that the copy range is specified by statement numbers instead of line numbers.

If the first parameter is "*", then all routines are copied to the specified file or device.

Notes: The /COPY command is the normal way of saving, in MTS files, routines which have been developed or debugged using IF. Of course, the routine may be in an MTS line file already, depending on the mode of compilation.

The /COPY command may be entered in "free-format"; the parentheses and the comma appearing in the prototype form above are optional.

Examples: /COPY CORN CORN.F

would copy all of the statements in the routine named CORN to the MTS file CORN.F.

/COPY GBAND(#66) GBANDSOURCE

would copy the statements of routine GBAND (starting at statement labeled 66 and continuing to the last statement in the routine) to the MTS file GBANDSOURCE.

October 1983

/DESTROY

Prototype: (a) /DESTROY routine ...

(b) /DESTROY *

Purpose: The /DESTROY command destroys routines compiled under IF.

Prototype (a) of the /DESTROY command destroys each routine specified in the parameter list. If the parameter is "*", all routines compiled under IF are destroyed.

Notes: The /DESTROY command has no effect on externally loaded routines, which can be unloaded using the /UNLOAD command.

An "invoked" routine may not be destroyed. This means that neither the active routine, nor any higher-level routine may be destroyed. For example, the routine which invoked the active routine may not be destroyed.

Destroying a routine is accomplished internally by simply deleting entries from the tables maintained by IF. Under no circumstance does destroying a routine have any effect on any MTS file.

Examples: /DESTROY SIMPLX DSIM

would cause the routines named SIMPLX and DSIM to be destroyed.

/DESTROY *

would destroy all routines compiled under IF except, of course, any invoked routines.

/DISPLAY

Prototype: /DISPLAY keyword ...

Purpose: Displays miscellaneous relevant information. "keyword" must be one of the following:

ACTIVE	- prints the name of the currently active routine
ATPOINTS	- prints a list of all currently defined atpoints in all routines
BREAKPOINTS	- prints a list of all currently defined breakpoints in all routines
COMMON	- prints a common storage map for the currently active routine
COST	- prints the accumulated cost of this job since $RUN *IF
CPUTIME	- prints the accumulated CPU time since $RUN *IF
ELTIME	- prints the accumulated elapsed time since $RUN *IF
ELAPSED	- same as ELTIME
EQUIVALENCE	- prints an equivalence storage map for the currently active routine
EXTERNAL	- prints a list of all externally loaded routines
LEVELS	- prints all points of suspension
LINE	- same as LEVELS
LOADED	- same as EXTERNAL
MEMORY	- prints the current size of the user's virtual memory as a decimal number of pages
NAME	- same as ACTIVE
PROGRAMS	- prints a list of all routines compiled by IF
ROUTINES	- same as PROGRAMS
STATUS	- prints status information including CPU-TIME, ELAPSED, VMSIZE, and COST
SUSPENSION	- same as LEVELS
VMSIZE	- same as MEMORY
XCSECTS	- same as EXTERNAL

Example: /DISPLAY ATPOINTS BREAKPOINTS

displays all currently defined breakpoints and atpoints in all routines.

October 1983

/EDIT

Prototype: (a) /EDIT routine

(b) /EDIT routine(linenumber)
/EDIT linenumber

(c) /EDIT routine(#label)
/EDIT #label

Purpose: The /EDIT command is used to enter edit mode explicitly so that source changes can be made to the source code corresponding to either a routine or an atpoint.

Prototype (a) of the /EDIT command invokes the editor on the routine specified in the parameter field. If the routine name is omitted, the currently active routine is assumed.

Prototypes (b) and (c) of this command invoke the editor on the atpoint which is defined at the specified statement. Once again, if the routine name is omitted, the currently active routine is assumed.

Notes: After the user has completed making his source changes in the editor, he may enter a null line or an editor RETURN command to return to IF and have recompilation automatically performed. To return to IF, but to bypass recompilation, the user should enter the editor command IF. If a routine contains a compilation error, then it may not be run until the compilation error has been corrected using the editor.

Examples: /EDIT MAIN1

would invoke the editor on the routine named MAIN1.

/EDIT MAIN1(37)

would invoke the editor on the atpoint defined at line 37.0 of routine MAIN1.

/EDIT OLQF(#120)

would invoke the editor on the atpoint defined at statement labeled 120 of routine OLQF.

/ERASE

Prototype: /ERASE

Purpose: This command can be issued in either immediate execution (prefix: *) or suspended execution (prefix: +).

When issued in immediate execution, the immediate execution environment is erased. This means that all variables, arrays, format statements, declarations, and so on currently defined in immediate execution disappear.

When issued in suspended execution, the suspended execution environment associated with the currently active routine is erased. Note that when execution is initiated by the /RUN command, the suspended execution environment associated with each routine is erased as part of the clearing process the first time the routine is referenced.

Example: The following example illustrates the effect of the /ERASE command in immediate execution:

```
* real i
* i=100
* print,i
      100.0000
* /erase
* print,i
  ERROR: I IS UNDEFINED
* i=100
* print,i
           100
*
```

October 1983

/EXECUTE

Prototype: /EXECUTE routine(linenumber1,linenumber2) count

Purpose: To execute a set of statements within a specified range. A suspension will result if an attempt is made to execute a statement outside the specified range; however, references to other routines are allowed.

If the routine name is omitted, the currently active routine is assumed. If "linenumber1" is omitted, the first executable statement in the routine is assumed. If "linenumber2" is omitted, the last executable statement in the routine is assumed.

If "count" is provided, then a suspension will occur after "count" FORTRAN statements have been executed. If "count" is not specified, it defaults to an infinite value.

Example: The following example illustrates the use of the /EXECUTE command:

```
* /compose
      1_ limit=2
  ROUTINE NAME: MAIN
      2_ do 10 i=1,limit
      3_ 10 print,i
      4_ /execute 1 3
             1
             2
      4_ stop
      5_ end
+ /immex
*
```

Note that the suspension which would normally occur after the execution of the /EXECUTE command is deferred until composing is completed.

/EXPLAIN

Prototype: (a) /EXPLAIN

(b) /EXPLAIN command ...

Purpose: If no parameter is given, a list of all the available IF commands together with a brief explanation of the command syntax is printed.

Prototype (b) provides information about the particular commands specified in the parameter list.

Note: The command prefix character (/) must appear even on commands specified on the /EXPLAIN command.

Example: /EXPLAIN /COMPOSE /RUN

would provide detailed information about how to use the /COMPOSE and /RUN commands.

October 1983

/GET

Prototype: /GET routine

Purpose: The specified routine becomes the currently active routine. This command has the effect of suspending at the first executable statement of the specified routine. Subsequently, the prefix character will be "+".

Notes: If a routine is already active when the /GET command is issued, then the existing suspension is stacked on a pushdown list before activating the specified routine and suspending on it. The levels of stacked suspension are printed by using the "/DISPLAY LEVELS" command. To delete the current level of suspension and to return to the previous level of suspension, the user should use the /RELEASE command. To delete all levels of suspension and return to immediate execution mode, the user should use the /IMMEX command. To return to immediate execution mode while preserving all levels of suspension, the user should issue a "/GET IMMEX" command.

The user should be aware that the /RUN command also has the effect of deleting all levels of suspension prior to beginning execution.

Example: Assume that the user is suspended at line 25.0 of routine MAIN, and that he wishes to activate routine TSP2 in order that he may easily examine the values of variables in that routine. To do this, he may issue the following command:

```
/GET TSP2
```

which makes TSP2 the currently active routine. Subsequently, by issuing the following command:

```
/RELEASE
```

routine MAIN would once again be the currently active routine, and he would still be suspended at line 25.

/HELP

Prototype: /HELP

Purpose: A command which prints information about how to terminate the current IF run, and about how to obtain information about the other IF commands.

Example:

```
* /help
*                  ***** IF *****
*
* IF YOU NEED SOME HELP WITH THE IF COMMANDS, THEN
* TYPE "/EXPLAIN".
*
* IF YOU WISH TO TERMINATE THIS *IF RUN, THEN TYPE
* "/STOP".
*
* IF YOU WISH ONLY TO RETURN TO MTS WITHOUT
* UNLOADING THE IF SYSTEM, THEN TYPE "/MTS" (AN MTS
* $RESTART COMMAND WILL CAUSE EXECUTION TO RESUME).
```

October 1983

/IMMEX

Prototype: /IMMEX

Purpose: To unconditionally return from suspended execution (prefix: +) to immediate execution (prefix: *).

Notes: Execution of the /IMMEX command causes all levels of suspension to be deleted. However, data environments are not cleared, and the values of the variables and array elements in any routine may still be examined.

The /IMMEX command has no effect when it is issued in immediate execution.

Example:

```
* /compose
      1_ print,aaa
  ROUTINE NAME: MAIN
      2_ return
      3_ end
* /run main
  /MAIN:1./ - ERROR:  AAA IS UNDEFINED
+ /immex
*
```

/INPUT

Prototype: /INPUT FDname

Purpose: IF will read subsequent immediate and suspended execution commands and FORTRAN source lines from the specified file or device. If an end-of-file is encountered on the specified input stream or if the attention key is pressed, subsequent commands and FORTRAN source lines will revert to being read from *MSOURCE* (the terminal in conversational mode). Initially, lines are read from *SOURCE*.

Example: /INPUT IFCMDS

would cause IF to read and execute commands from the MTS file IFCMDS.

October 1983

/LIBRARY

Prototype: (a) /LIBRARY

(b) /LIBRARY FDname

Purpose: To define a library or libraries for IF to search when references to undefined routines are encountered during program execution. By default, only the public library, *LIBRARY, is searched.

Prototype (a) of this command indicates that no libraries are to be searched. When a reference to an undefined routine is encountered during program execution, IF will suspend with an error message like,

ERROR: ROUTINE "xxxxxx" IS UNDEFINED

and the user will find himself in either immediate execution mode (prefix: *) or suspended execution mode (prefix: +).

Prototype (b) of this command dictates that whenever an undefined routine is referenced, IF will search the specified library for the undefined symbol. More than one library may be provided by concatenating the names of libraries. Note that prototype (b) of this command also implies that *LIBRARY is automatically searched, but only after the programmer's libraries have been searched.

Example: /LIBRARY NUMLIB+NEWLIB

would cause IF to search the libraries residing in the MTS files NUMLIB, NEWLIB, and *LIBRARY (in that order) whenever an undefined routine was referenced during program execution.

/LINK

Prototype: /LINK FDname I/O-assignments PAR=string

Purpose: Load and begin execution of the object program in the specified file without leaving IF.

Logical I/O Unit Assignments

The following table lists the I/O units which can be assigned on the /LINK command, and their defaults if they are not assigned:

0-4	do not default
5	*SOURCE*
6	*SINK*
7-19	do not default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH*, if in batch mode
GUSER	*MSOURCE*
SERCOM	*MSINK*

Note: The /LINK command is useful because it allows the programmer to load and run any program without unloading IF. After the program which was /LINKed to has completed execution, control will be returned to IF -- providing the program /LINKed to exits normally.

Example: The following example shows how a program residing in the MTS file TRIP.S may be compiled using the FORTRAN-G compiler, and how the object module produced may be loaded under IF:

```
* /link *ftn scards=trip.s
   NO ERRORS IN TRIP
* /load -load
*
```

October 1983

/LIST

Prototype: (a) /LIST routine(linenumber1,linenumber2) FDname

(b) /LIST routine(#label1,#label2) FDname

(c) /LIST * FDname

Purpose: Prototype (a) of this command lists a routine, or a range of lines of a routine, on the specified file or device. This command is very similar to the MTS $LIST command. If the routine name is omitted, the currently active routine is assumed. If "linenumber1" is omitted, the first statement in the routine is assumed. If "linenumber2" is omitted, the last statement in the routine is assumed. If "FDname" is omitted, the current output file or device is assumed. The current output file or device, which can be redefined by the /OUTPUT command, defaults to *MSINK*.

Prototype (b) is similar to prototype (a) except that the list range is specified by statement numbers instead of line numbers.

If the first parameter is "*", then all routines are listed on the specified file or device.

Note: The /LIST command may be entered in "free-format"; the parentheses and comma in the prototype form above are optional.

Examples: /LIST MAIN

would list all of the statements in the routine named MAIN.

/LIST EVAL (#66)

would list the statements of routine EVAL starting at statement labeled 66 and continuing to the last statement in the routine.

/LIST * *PRINT*

would list all routines on a printer.

/LOAD

Prototype: /LOAD FDname

Purpose: The /LOAD command dynamically loads all object modules residing on the specified file or device. The loaded object modules or csects are subsequently callable from any routine compiled under IF, and conversely may call any routine compiled under IF.

Notes: The word "csect" is a synonym for "externally loaded module" or "externally loaded routine".

To obtain a list of all currently loaded csects, the user should issue the "/DISPLAY EXTERNAL" command.

To unload all csects, or to unload a particular csect, the user should issue the /UNLOAD command.

If the specified file or device name is a library, then only modules with outstanding references are loaded from the library. Note that IF automatically searches the public library, *LIBRARY, for undefined symbols. To have IF automatically search a programmer-supplied library, see the /LIBRARY command.

Example: The following example illustrates how the FORTRAN-G compiler may be invoked to compile a FORTRAN source program residing in the MTS file ATOMPGM.S. Subsequently, the object module generated by *FTN is loaded by using the /LOAD command, and the /DISPLAY command is used to list the names of the loaded csects:

```
* /link *ftn scards=atompgm.s spunch=atompgm.o
  NO ERRORS IN ATOMB
  NO ERRORS IN HEAVYW
* /load atompgm.o
* /display external
* EXTERNALLY LOADED CSECTS:
* ATOMB ADDRESS=518000 LENGTH=2F20
* HEAVYW ADDRESS=521000 LENGTH=550
*
```

October 1983

/MTS

Prototype: (a) /MTS

(b) /MTS MTS-command

Purpose: Prototype (a) of the /MTS command returns control to MTS command mode. IF processing may be resumed by issuing the MTS command $RESTART.

Prototype (b) of the /MTS command executes the specified MTS command in the parameter field by calling the system subroutine CMD. In this case, control automatically returns to IF after the command is executed (unless the MTS command is a $LOAD, $RUN, $RERUN, $UNLOAD, or $DEBUG command).

Note: It is not necessary to return to MTS command mode to assign or reassign logical I/O units. With IF commands there are two ways to assign logical I/O units. Issue the /SET command (i.e., "/SET 1=-SCRATCH1"), or assign the units on the /RUN command (for example, "/RUN MAIN 6=OUT").

Example: /MTS $LIST SUB.S(1,50)

lists lines 1-50 of the MTS file named SUB.S without returning to MTS command mode.

/OUTPUT

Prototype: /OUTPUT FDname

Purpose: IF will direct subsequent output lines to the specified file or device. This does not include output produced by the programmer's routines, or error messages produced by IF which are always written on *MSINK*. It does include output like routine listings, and other "less important" information. Initially, output lines are printed on *MSINK* (the terminal, in conversational mode).

Example: /OUTPUT -OUT

would cause IF to direct output to the MTS file named -OUT.

October 1983

/REFERENCE

Prototype: (a) /REFERENCE routine:variable ...
/REFERENCE variable ...

(b) /REFERENCE routine:#label ...
/REFERENCE #label ...

Purpose: This command prints the line numbers of the statements at which the variable names and the statement labels specified in the parameter list are referenced. If the routine name is omitted, the currently active routine is assumed.

Examples: /REFERENCE LOOPI MAIN5:BALL

would print the references of the variable LOOPI in the currently active routine, and the variable BALL in routine MAIN5. The references are printed in the following format:

```
"LOOPI" REFERENCED AT LINES:
140. 209. 224. 241. 250.

"MAIN5:BALL" REFERENCED AT LINES:
12. 13.5 18. 19. 55. 56.1
```

/REFERENCE #1234

would print the references of statement label 1234 in the currently active routine in the following format:

```
"#1234" DEFINED AT LINE 10.,
REFERENCED AT LINES:
10. 88. 89. 101.
```

/RELEASE

Prototype: /RELEASE

Purpose: Deletes the current level of suspension, and returns to the previous level of suspension. If there is only one level of suspension, then this results in a return to immediate execution mode (prefix: *).

Notes: The levels of stacked suspension are printed by using the "/DISPLAY LEVELS" command.

To delete all levels of suspension and return to immediate execution mode, issue the /IMMEX command.

Example: The following example illustrates the use of the /RELEASE command,

```
* /run main
  /MAIN:3./ - ERROR:  J IS UNDEFINED
+ /display levels
+ SUSPENSION AT LINE 3. OF ROUTINE MAIN
+ /get subr
  /SUBR:1./
+ /display levels
+ SUSPENSION AT LINE 1. OF ROUTINE SUBR
+ SUSPENSION AT LINE 3. OF ROUTINE MAIN
+ /release
+ /display levels
+ SUSPENSION AT LINE 3. OF ROUTINE MAIN
+ /release
*
```

October 1983

/REMOVE

Prototype: (a) /REMOVE

(b) /REMOVE routine(linenumber) ...
/REMOVE linenumber ...

(c) /REMOVE *

Purpose: Prototype (a) of this command removes the most recently executed atpoint or breakpoint.

Prototype (b) of this command removes the atpoints or breakpoints specified in the parameter list. If the routine name is omitted, the currently active routine is assumed. If the line number is omitted, the first executable statement of the specified routine is assumed.

If the parameter is "*", then all atpoints and breakpoints in all routines are removed.

Notes: The "/DISPLAY BREAKPOINTS" command prints a list of all currently defined breakpoints, and the "/DISPLAY ATPOINTS" command prints a list of all currently defined atpoints.

If both an atpoint and a breakpoint are defined at the same statement, then the breakpoint is removed.

By using the "/SET BREAK=OFF" command, breakpoints may be disabled globally without removing them. Atpoints may be disabled in the same way by issuing the "/SET AT=OFF" command.

Examples: /REMOVE 123.4

would cause the atpoint or breakpoint defined at line 123.4 of the currently active routine to be removed.

/REMOVE MAIN(11) MAIN(#8888)

would cause the atpoints or breakpoints defined at line 11.0 of routine MAIN, and at statement labeled 8888 of routine MAIN to be removed.

/REPEAT

Prototype: /REPEAT

Purpose: Restarts execution of a suspended program by repeating the statement at which the suspension occurred. (See also the descriptions of the /CONTINUE and /RESTART statements.)

Note: The current statement, that is the statement at which execution is suspended, can be found by using the "/DISPLAY LEVELS" command.

Example: The following example illustrates the use of the /REPEAT command:

```
* /compose
      1_ do 10 i=1,n
  ROUTINE NAME: MAIN
      2_ 10 print,i
      3_ stop
      4_ end
* /run main
  /MAIN:1./ - ERROR:  UNDEFINED DO PARAMETER N
+ n=2
+ /repeat
              1
              2
  /MAIN:3./ - /STOP /
+
```

October 1983

/RESTART

Prototype: (a) /RESTART

(b) /RESTART routine(linenumber)
/RESTART linenumber ...

(c) /RESTART routine(#label) ...
/RESTART #label ...

Purpose: Restarts execution of a suspended program. Prototype (a) of this command causes execution to restart in the statement at which the suspension occurred. (See also the descriptions of the /CONTINUE and /REPEAT statements.)

Prototypes (b) or (c) of this command explicitly specify the routine and statement at which execution is to restart. If the routine name is omitted, the currently active routine is assumed. If the line number is omitted, the first executable statement of the specified routine is assumed.

Notes: The current statement, that is the statement at which execution is suspended, can be found by using the "/DISPLAY LEVELS" command.

After a suspension caused by an external interrupt (i.e., after either a program interrupt or an attention interrupt in an externally loaded module) the first form of the /RESTART command has a special interpretation. For this type of suspension only, execution is restarted within the externally loaded module at the next instruction.

Examples: /RESTART MAIN(24.5)

would restart execution at line 24.5 of routine MAIN.

/RESTART #8888

would restart execution at statement labeled 8888 in the currently active routine.

/RUN

Prototype: /RUN routine I/O-assignments PAR=string

Purpose: Begins execution of the specified routine. The /RUN command is a typical method of beginning execution of a mainline program, or of a subroutine without arguments.

Logical I/O Unit Assignments

All logical unit assignments other than those appearing on the /RUN command become unassigned every time the /RUN command is issued. The following table lists the I/O units which can be assigned on the /RUN command, and their defaults if they are not assigned:

0-4	do not default
5	*SOURCE*
6	*SINK*
7-19	do not default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH*, if in batch mode
GUSER	*MSOURCE*
SERCOM	*MSINK*

The PAR Field

If the PAR field is specified, then it must be the last field on the /RUN command. Any fields following the PAR= are assumed to be part of the PAR text. The PAR field defaults to being a field of length zero.

Notes: When execution is initiated by the /RUN command, the data environment associated with each routine which has been compiled under IF is implicitly "cleared" the first time it is invoked. Clearing a routine means setting all variables and arrays in the routine to being undefined, and then assigning initial data values to those variables and arrays which are given initial data values in DATA and explicit type statements.

When execution is initiated by the /RUN command, all levels of suspension are automatically deleted.

The /RUN command can also be used to begin execution of any externally loaded routine (csect).

Examples: /RUN MAIN 5=LPDATA

would attach logical unit 5 to the MTS file LPDATA, and then begin execution of the routine named MAIN.

```
* /load lip.o
* /display external
  LIP ADDRESS=508400 LENGTH=218
  LIPSUB ADDRESS=50A100 LENGTH=488
* /run lip par=notab
```

loads the external routine in the MTS file LIP.O, and transfers control to the entry point of the loaded object module called LIP.

/SET

Prototype: /SET keyword=rhs ...

Purpose: The /SET command is used to assign or reassign logical I/O units, and to set various global switches which control the behavior of IF.

Assigning and Reassigning Logical Units

Normally, MTS logical units are assigned on the /RUN command. However, sometimes it is desirable to change logical units halfway through a run, without having to /RUN the routine from the beginning. This can be done by issuing the /SET command from suspended execution (e.g., /SET 5=DATAFILE). The following table lists the units which can be assigned, and their defaults if they are not assigned:

0-4	do not default
5	*SOURCE*
6	*SINK*
7-19	do not default
SCARDS	*SOURCE*
SPRINT	*SINK*
SPUNCH	*PUNCH*, if in batch mode
GUSER	*MSOURCE*
SERCOM	*MSINK*

Assigning logical units using the /SET command is equivalent to returning to MTS and assigning the logical units on an MTS $RESTART command.

Other Functions of the /SET Command

AT={ON|OFF} Default: ON

If this switch is OFF, then atpoints encountered in the execution flow are ignored.

BREAK={ON|OFF} Default: ON

If this switch is OFF, then breakpoints encountered in the execution flow are ignored.

CMDCHAR=character Default: /

The command prefix character becomes the character specified by "character". "character" must be chosen from the set {¢<&!*>?#@'=$/}.

CMTCHAR=character Default: "

The free-format comment character becomes the character specified by "character". "character" must be chosen from the set {¢<&!*>?#@'=}.

CONTCHAR=character Default: -

The free-format continuation character becomes the character specified by "character". "character" must be chosen from the set {¢<&!*>?#@'=-}.

DEFCHK={ON|OFF} Default: ON

If this switch is OFF, then the checks for undefined variables and array elements in FORTRAN output statements are disabled.

ECHO={ON|OFF}

The function of this switch is to turn echo printing ON or OFF. When echoing is ON, all FORTRAN source lines and commands read from a nonterminal file or device are echoed on the terminal (the printer in batch mode). In batch mode, echoing defaults to ON. In conversational mode, echoing defaults to OFF.

FLOW={ON|OFF} Default: OFF

If this switch is ON, then each time a branch is made, the routine name and label are printed. This is a very useful feature. Sections of programs can be flow traced by setting this switch ON or OFF within atpoints.

LC={ON|OFF} Default: OFF

If this switch is OFF, FORTRAN source lines and commands are automatically translated to uppercase. This can also be accomplished by specifying /SET UC=ON.

LENGTH=count Default: 72

"count" must be an integer between 36 and 72 inclusive. Free-format FORTRAN source statements longer than "count" characters are automatically broken up into the appropriate number of continuation lines, with no line in the statement being longer than "count" characters.

LENCHK={ON|OFF} Default: ON

If this switch is ON, then fixed format FORTRAN source lines longer than 72 characters will be flagged with warning messages. This is not a desirable action when one wishes to compile a program that has sequence fields.

MAP={ON|OFF} Default: OFF

If this switch is ON, a loader map will be printed whenever IF either explicitly or implicitly loads object modules (such as compiled by *FTN).

MSGFILE={ON|OFF} Default: OFF

If this switch is ON, then the IF error message file is released after each error message is printed. In the OFF position, the error message file remains open between error messages. In the ON position approximately four pages of virtual memory are saved; in the OFF position, a small amount of CPU time is saved (i.e., the CPU time required to open and close the file).

MSGLVL={0|1|2|3} Default: 1

This controls the verbosity level of error and warning messages. There are four ascending levels. The most terse level is 0; the most verbose level is 3.

PAR=string

This defines or redefines the PAR field. If PAR is specified, it must be the last keyword on the command. The default PAR field is a field of length zero.

UC={ON|OFF} Default: ON

If this switch is ON, FORTRAN source lines and commands are automatically translated to uppercase. This can also be accomplished by specifying /SET LC=OFF.

WARN={ON|OFF} Default: ON

If this switch is OFF, then all warning messages are suppressed.

Examples: /SET 5=-INPUT 6=RESULTS

assigns unit 5 to the MTS file -INPUT, and unit 6 to the MTS file RESULTS.

/SET WARN=OFF

suppresses all warning messages.

/STEP

Prototype: (a) /STEP

(b) /STEP count

Purpose: Execution resumes at the current point of suspension. The specified number of FORTRAN statements are executed, after which execution suspends.

If no parameter is given, the /STEP command executes one FORTRAN statement and suspends. If an integer count is given, IF executes that number of FORTRAN statements and suspends.

Note: The /STEP command executes "count" executable FORTRAN statements, not "count" machine instructions. A reference to an elementary FORTRAN function, and a call to an externally loaded routine both count as only one FORTRAN statement. Furthermore, when an atpoint is encountered during program execution, the statements in the atpoint are not counted.

Example: The following example illustrates the action of the /STEP command,

```
* /compose
      1_ print,'hi there'
  ROUTINE NAME: MAIN
      2_ print,'mom and dad'
      3_ return;end
* /get main
  /MAIN:1./
+ /step
  HI THERE
  /MAIN:2./
+ /step
  MOM AND DAD
  /MAIN:3./
+ /step
*
```

/STOP

Prototype: /STOP

Purpose: IF processing is terminated, and control is returned to MTS command mode. Execution may not be restarted. This is the normal method of terminating an IF run.

Notes: Some of the other ways of returning to MTS command mode are issuing the /MTS command or issuing a FORTRAN STOP statement from immediate execution mode (in these cases execution may be restarted using the MTS command $RESTART).

| Example:

```
# $run *if66
# EXECUTION BEGINS
* IF(NOV80)
* " it worked
* /stop
# EXECUTION TERMINATED
#
```

Page Revised February 1988 October 1983

/TRACE

Prototype: /TRACE

Purpose: To produce a subprogram linkage traceback.

Example: The following example illustrates the value of the /TRACE command:

```
* /run main
   /SUBP:99./ - ERROR:  VAR8 IS UNDEFINED
+ /trace
  CALLED FROM ROUTINE "PRINTR", STATEMENT 50.5
  CALLED FROM ROUTINE "LASTS", STATEMENT 18
  CALLED FROM ROUTINE "MAIN", STATEMENT 5
  INITIATED FROM IMMEDIATE EXECUTION MODE
+
```

October 1983

/UNLOAD

Prototype: (a) /UNLOAD

(b) /UNLOAD csect ...

(c) /UNLOAD *

Purpose: The /UNLOAD command dynamically unloads selected external routines. The routines unloaded may have been implicitly loaded from a library, or explicitly loaded using the /LOAD command. Subsequent references to an unloaded routine either from another externally loaded routine, or from a routine compiled under IF, will behave as if the routine had never been loaded.

Prototype (a) of this command unloads the most recently loaded external routine.

Prototype (b) unloads each of the external routines specified in the parameter list.

Prototype (c) unloads all currently loaded external routines.

Notes: The word "csect" is a synonym for "externally loaded routine." To obtain a list of all currently loaded csects issue the "/DISPLAY EXTERNAL" command. This will produce a list of loaded external csects, ordered starting with the most recently loaded.

When an externally routine is unloaded, all entry points which were in the routine (if any) also become undefined.

External block data may be unloaded by /UNLOADing the name of each common block for which block data exists.

Example: To unload all externally loaded csects, enter the "/UNLOAD *" command as follows;

```
* /unload *
    "SPSSUB" UNLOADED
    "WRITER" UNLOADED
*
```

/WORKFILE

Prototype: /WORKFILE filename

Purpose: Compilations initiated either by the /COMPOSE command, or the first or third forms of the /COMPILE command, cause statements being compiled to be copied automatically to the MTS file specified by "filename". This file is known as the workfile. While the workfile is for the most part transparent to the programmer, the programmer should be aware that subsequent processing (i.e., editing, listing, copying) will reference only the statements maintained on the workfile. If the /WORKFILE command is not issued, then IF automatically uses a scratch workfile named -FSFILE. The workfile is emptied by IF prior to the first compilation.

Notes: The specified file must be a line file, and the user must have both read and write access to it.

While it is usually not necessary, the user may, if he wishes, use as many workfiles as desired. The /WORKFILE command overrides the effect of any previous /WORKFILE command.

The /WORKFILE command is sometimes useful when one wishes to run IF for a second time during the same terminal session. As an example, imagine that a programmer composed a series of routines on a first IF run, and that the source statements for these routines were copied to the default workfile (-FSFILE) by IF. Imagine further that on a second IF run the programmer wished to recompile the same routines he composed during the first run. To do this, he could issue the /WORKFILE command specifying an alternate workfile (any file but -FSFILE), and then issue the /COMPILE command to compile the routines directly from the previous workfile (-FSFILE) as follows:

```
* /workfile -w
* /compile from -fsfile
  ROUTINE NAME: MAIN
  ROUTINE NAME: ROOT
*
```

If the programmer did not issue the /WORKFILE command before he issued the /COMPILE command, then IF would have used -FSFILE as the workfile on the second run. In this case, the routines would have been lost as IF would have emptied -FSFILE prior to the beginning of the compilation.

Example:

```
* /workfile -w
  ERROR: "-W" IS NOT EMPTY; REPLY "OK" TO EMPTY
? ok
*
```

APPENDIX B: LANGUAGE FEATURES SUPPORTED

| The *IF66 version of the IF system supports the language described in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form number GC28-6515, except for the following restrictions and extensions.

Multiple-Assignment Statements

Multiple-assignment statements are accepted by the IF system as an extension to the FORTRAN language. They are not accepted by the FORTRAN-G or FORTRAN-H compilers. For example:

```
A=B(3)=DD= 1.234
```

Free-Format I/O

The IF system supports the same free-format I/O statements as the WATFIV compiler supports. The following statements may be used to achieve free-format I/O:

```
READ, list
PRINT, list
PUNCH, list
READ(unit,*,END=label1,ERR=label2) list
WRITE(unit,*) list
```

I/O in the first three forms above is done on units 5, 6, and 7, respectively. The asterisks in the last two forms imply free-format I/O with "unit" specifying the logical unit number. The END and ERR fields are optional as with the standard FORTRAN READ statement. For example:

```
READ, A(1),B,I
2.5,.00001 13
```

would assign the value 2.5 to A(1), .00001 to B, and 13 to I.

Input data items may be entered on as many lines as necessary; successive lines are read until all elements of the data list have been satisfied. Adjacent data fields must be separated by a comma and/or one or more blanks. Individual data fields must be wholly contained on a single line--they cannot be continued across two lines.

Free-format output data items are printed using the following predefined format codes:

```
INTEGER     I12
REAL*4      G16.7
REAL*8      G28.16
COMPLEX*8   '(',G16.7,',',G16.7,')'
COMPLEX*16  '(',G28.16,',',G28.16,')'
LOGICAL     L8
```

For example,

```
PRINT,'I=',I
I=          13
```

More than one line of output is printed if the entire data list will not fit onto a single line.

Notice that the IF system also supports expressions in output lists. For example,

```
PRINT,'RESULT=',1*2+A/B
RESULT=    2.500000
```

Implied DO-Loops in DATA Statements

The IF system supports implied DO loops in DATA statements. For example,

```
REAL A(10)
DATA (A(I),I=1,10)/10*1.0/
```

Extended Ranges on DO-Loops

The IF system does not support the concept of the "extended range of the DO loop" (such as described in the above named IBM manual). Furthermore, the value of the DO variable or DO index upon completion of a DO loop is undefined when running under IF (for the FORTRAN-G, FORTRAN-H, and WATFIV compilers the DO index remains defined but the value of the index varies--contrary to ANSI FORTRAN specifications).

Debug Facility

The IF system does not support the FORTRAN debug facility; thus, one cannot use IF to compile any FORTRAN programs containing the DEBUG, AT, TRACE, or DISPLAY debug statements. Simply remove debug statements from FORTRAN programs (or change them into comments).

Call by Location

The IF system has implemented subroutine "calls by location" as subroutine "calls by value." Thus, with the IF system, there is no difference in interpretation between the following two FORTRAN statements:

```
SUBROUTINE SUB(A,/B/)
SUBROUTINE SUB(A,B)
```

Predefined Functions

The elementary FORTRAN functions are "pretyped" when using the IF system. For example, the double-precision square root function (DSQRT) is pretyped to be REAL*8. Functions are "pretyped" using the FORTRAN-G and FORTRAN-H compilers, but not the WATFIV compiler.

Comments

Although ANSI Standard FORTRAN prohibits comment lines interspersed with continuation lines and the abovementioned IBM manuals prohibit such interspersing, the FORTRAN-G and FORTRAN-H compilers do not enforce this restriction. IF does not permit the interspersing of comments and continuation lines.

NAMED COMMON Restriction

IF does not permit a COMMON block to be named MAIN. FORTRAN-G and FORTRAN-H do not have this restriction.

Declaration of Dimensioning Restriction

IF requires that an array be dimensioned before it is referenced by another declaration. The following is not permitted in IF:

```
EQUIVALENCE X, Y(3)
DIMENSION Y(5)
```

The DIMENSION statement must precede the EQUIVALENCE statement.

APPENDIX C: DETAILED EXAMPLES

This appendix contains two detailed examples of conversational usage of the IF system. Text appearing in lowercase was entered by the programmer, and text appearing in uppercase was printed either by MTS or by the IF system.

Example 1

In this example, the programmer signs on at a terminal and writes a FORTRAN program which reads in a series of real numbers, sorts them in ascending order, and prints out the sorted list. An explanation of the steps follows the example.

```
      # $sig ifdo
      # ENTER USER PASSWORD.
      ? xxxxxx
      # TERM,NORMAL,UNIV
      # **LAST SIGNON WAS: 13:37:16
      #   USER "IFDO" SIGNED ON AT  16:41:05 ON WED FEB 04/76
|     # $run *if66                                              ...(1)
      # EXECUTION BEGINS
|     * IF(NOV80)
      * /compose                                                ...(2)
           1_ " program to do a simple sort
           2_ dimension array(100)
        ROUTINE NAME: MAIN                                      ...(3)
           3_ " first read in the unsorted list
           4_ n=0
           5_ do 20 i=1,100
           6_ read(5,10,end=30)array(i)
           7_ 10 format(i10)
           8_ n=n+1
           9_ 20 continue
          10_ " now sort the list
          11_ 30 nm1=n-1
          12_ do 50 i=1,nm1
          13_ ip1=i+1
          14_ do 40 j=ip1,n
          15_ if(array(i).le.array(j))goto 20
        ERROR: ATTEMPT TO TRANSFER TO 20 WHERE 20 IS IN         ...(4)
               THE RANGE OF A DO. TRANSFER FROM LINE 15
               TO LINE 9; WITHIN "DO 20", RANGE=(5,9)
      : change #20#30#                                          ...(5)
      :      15.          IF(ARRAY(I).LE.ARRAY(J))GOTO 30
      : stop                                                    ...(6)
          16_ itemp=array(i)                                    ...(7)
          17_ array(i)=array(j)
          18_ array(j)=itemp
          19_ 40 continue
          20_ 50 continue
          21_ " now print the sorted list
```

```
      22_ do 60 i=1,n
      23_ write(2,10)array(i)
      24_ 60 continue
      25_ stop
      26_ end
* /run main                                                    ...(8)
  1.0                                                         ...(9)
  ***ERROR*** FORMAT CODE AND VARIABLE TYPE DO NOT            ...(10)
  MATCH. FORMAT CODE IS I10, TYPE IS REAL. CONDITION
  OCCURRED DURING A FORMATTED READ ON FORTRAN UNIT
  5 WHICH IS ATTACHED TO *SOURCE*.
  /MAIN:6./                                                   ...(11)
+ /edit                                                       ...(12)
: line 7
:      7.       10 FORMAT(I10)
: change #i10#f10.0#
:      7.       10 FORMAT(F10.0)
: stop
+ /restart                                                    ...(13)
  1.0
  3.0
  2.0
  $endfile
  (attention key pressed)                                     ...(14)
  /MAIN:13./ - ***** ATTENTION INTERRUPT                      ...(15)
+ /list 13 13                                                 ...(16)
+    13.          IP1=I+1
+ nm1,n                                                       ...(17)
+            2           3
+ /step                                                       ...(18)
  /MAIN:14./                                                  ...(19)
+ /step
  /MAIN:15./
+ /step
  /MAIN:11./                                                  ...(20)
+ /edit                                                       ...(21)
: line 15
:     15.          IF(ARRAY(I).LE.ARRAY(J))GOTO 30
: change #30#40#
:     15.          IF(ARRAY(I).LE.ARRAY(J))GOTO 40
: stop
+ /run                                                        ...(22)
  1.0
  3.0
  2.0
  $endfile
  ***ERROR*** UNIT WAS REFERENCED BUT WAS NOT                 ...(23)
  ASSIGNED OR DEFAULTED. CONDITION OCCURRED DURING
  A FORMATTED WRITE ON FORTRAN UNIT 2.
  2 WAS CALLED BUT NOT SPECIFIED.
  ENTER NAME OR "CANCEL".
  *sink*                                                      ...(24)
        1.
```

```
          2.
          3.
    /MAIN:25./ - /STOP/                                  ...(25)
  + /mts $create sort.s                                  ...(26)
  # $CREATE SORT.S
  #  FILE "SORT.S" HAS BEEN CREATED.
  + /copy main sort.s                                    ...(27)
  + /stop                                                ...(28)
  # EXECUTION TERMINATED
```

Explanation of Example 1

(1) The programmer signs on and invokes the IF system with the MTS command "$RUN *IF".
(2) The /COMPOSE command is entered in order to begin compiling free-format statements. Notice that this form of the /COMPOSE command causes automatic numbering of input lines (like the $NUMBER command in MTS).
(3) The line "ROUTINE NAME: MAIN" is printed by the IF system. After the first statement of a routine is compiled the routine name is printed.
(4) A compilation error is detected and an error message is produced. The editor is invoked. Notice that the prefix character becomes a colon (:), indicating edit mode.
(5) The editor CHANGE command is used to change the statement label 20 to 30.
(6) The editor STOP command is entered to terminate the editing process. Recompilation of the line changed in the editor will be automatically performed.
(7) The recompilation was successful, and compilation resumes normally. The IF system prompts the programmer to enter line 16.
(8) The /RUN command is used to begin execution of the main routine which has just been compiled.
(9) This is an input data line.
(10) An execution error is detected by the FORTRAN I/O interface.
(11) Because of the I/O error execution suspends. The information which is printed between the slashes indicates that the error occurred at line 6 of routine MAIN. Notice that the prefix character becomes a plus sign (+), indicating suspended execution mode.
(12) The /EDIT command is entered to explicitly edit the active routine. In the editor the erroneous I10 format code is changed to a F10.0 format code.
(13) Having returned from the editor--back to suspended execution, the /RESTART command is issued. This causes execution to restart, beginning with the statement at which the error occurred (the READ statement).
(14) About 15 seconds passed here. It became apparent to the programmer that his program was in a loop, which required that he press the attention key.
(15) The attention interrupt is acknowledged by the IF system, and

execution suspends at line 13.

(16) The /LIST command is used to view statement 13.

(17) An expression statement is entered NM1,N to display the values of variables NM1 and N. These values seem reasonable.

(18) The programmer decides to use the /STEP command to step through his program one statement at a time. He is trying to determine why his program is looping (i.e., which statements are in the loop).

(19) The /STEP command executes one statement, and suspends at the next statement.

(20) Here the reason for the loop becomes clear to the programmer. The GOTO 30 statement at line 15 should have been GOTO 40.

(21) Once again the /EDIT command is used to enter edit mode.

(22) Having returned from the editor, the /RUN command is entered, causing the program to be rerun from the beginning.

(23) Another error is detected by the FORTRAN I/O interface. In this case the programmer forgot to assign unit 2. He should have done so on the /RUN command ("/RUN MAIN 2=*SINK*").

(24) Unit 2 is assigned to the terminal (*SINK*).

(25) The program executed correctly; the list of input data has been sorted into ascending order. Here execution suspends as the STOP statement is executed.

(26) The /MTS command is used to execute the MTS "$CREATE SORT.S" command. Here the intention is to create a permanent file in which to save the routine that has just been compiled.

(27) The /COPY command is used to save the routine MAIN in the MTS file named SORT.S.

(28) The /STOP command is entered to terminate the IF run.

Example 2

In this example, the programmer's project is to convert the sorting program which he compiled in the previous example into a subroutine. The program which was compiled in the previous example reads in a series of real numbers, sorts them into ascending order, and prints out the sorted list. The programmer's task is to convert this main program into a subroutine which is passed a list of real numbers as an argument (rather than reading in an unsorted list), and which returns a sorted list (rather than printing out a sorted list).

```
|      # $run *if66                                          ...(1)
       # EXECUTION BEGINS
|      * IF(NOV80)
       * /compile from sort.s                               ...(2)
         ROUTINE NAME: MAIN
       * /display routines                                  ...(3)
       * "MAIN"  RANGE=( 1. , 26. ) TYPE=MAIN
       * /edit main                                         ...(4)
       : insert 1 #subroutine sort(array,n)#
       : change 2 #100#n#
       :       2.          DIMENSION ARRAY(N)
       : delete 3 9
```

```
: delete 21 24
: replace 25 'return'
:      25.          RETURN
: stop
  ROUTINE NAME: SORT                                          ...(5)
* /list sort                                                  ...(6)
*     1.    C     PROGRAM TO DO A SIMPLE SORT
*     1.25        SUBROUTINE SORT(ARRAY,N)
*     2.          DIMENSION ARRAY(N)
*    10.    C     NOW SORT THE LIST
*    11.       30 NM1=N-1
*    12.          DO 50 I=1,NM1
*    13.          IP1=I+1
*    14.          DO 40 J=IP1,N
*    15.          IF(ARRAY(I).LE.ARRAY(J))GOTO 40
*    16.          ITEMP=ARRAY(I)
*    17.          ARRAY(I)=ARRAY(J)
*    18.          ARRAY(J)=ITEMP
*    19.       40 CONTINUE
*    20.       50 CONTINUE
*    25.          RETURN
*    26.          END
* dimension a(3)                                             ...(7)
* a(1)=1.0; a(2)=3.0; a(3)=2.0                               ...(8)
* call sort(a,3)                                             ...(9)
* a                                                          ...(10)
      1.000000         2.000000         3.000000
* /mts $create sortsub.s                                     ...(11)
# $CREATE SORTSUB.S
#  FILE "SORTSUB.S" HAS BEEN CREATED.
* /copy sort sortsub.s                                       ...(12)
* /link *ftn par=source=sortsub.s                            ...(13)
  NO ERRORS IN SORT
* /load -load                                                ...(14)
* WARNING: SYMBOL "SORT" IS MULTIPLY DEFINED;
*          FIRST DEFINITION USED.
* /destroy sort                                              ...(15)
*  "SORT" DESTROYED
* /load -load                                                ...(16)
* /display external                                          ...(17)
* EXTERNALLY LOADED CSECTS:
* SORT ADDRESS=50D450 LENGTH=438
* a(1)=3.0; a(2)=2.0; a(3)=1.0                               ...(18)
* call sort(a,3)                                             ...(19)
* a                                                          ...(20)
      1.000000         2.000000         3.000000
* /stop                                                      ...(21)
# EXECUTION TERMINATED
```

Explanation of Example 2

(1) The programmer invokes the IF system, again with the MTS command "$RUN *IF66".

(2) The /COMPILE command is entered, causing the sorting program which was compiled in the previous example to be compiled from the MTS file named SORT.S. The line which reads "ROUTINE NAME: MAIN" is confirmation that a main routine was compiled.
(3) The /DISPLAY ROUTINES command is used to have information printed concerning the routine which was just compiled.
(4) The /EDIT command is used to enter edit mode. In the editor the programmer makes the necessary modifications to transform his main program to a subroutine.
(5) This line was printed by the IF system during recompilation to inform the programmer that indeed the routine name has changed (from MAIN to SORT).
(6) The /LIST command is issued, causing the converted routine to be listed.
(7) Now the programmer is at the stage where he wants to test his subroutine to ensure that his editing was correct. The easiest way to do this is simply to invoke the subroutine from immediate execution mode. Here he declares a real array of dimension 3.
(8) And here a multiple statement line is used to assign values to the array elements.
(9) And here a CALL statement is used to invoke the subroutine.
(10) Control has returned from the subroutine; an expression statement is used to have the array A printed. The subroutine executed correctly--as one can see, the array was sorted in ascending order.
(11) The /MTS command is used to execute the MTS "$CREATE SORTSUB.S" command. The intention is to create a permanent file in which to save the subroutine that has just been debugged.
(12) The /COPY command is used to save the subroutine SORT in the MTS file named SORTSUB.S.
(13) At this point the programmer has decided to try compiling his sorting subroutine by using the *FTN compiler. He issues the /LINK command to invoke *FTN and have it compile the subroutine which is now in the file named SORTSUB.S.
(14) Here the programmer loads the object module which was produced by *FTN. The load fails because there is already a routine named SORT defined in the IF system.
(15) To get around this problem, the /DESTROY command is issued to destroy the routine which was compiled under IF at step (2).
(16) Now the /LOAD command is entered in an attempt to load the object module again. The load was successful because no diagnostic messages were produced.
(17) The /DISPLAY EXTERNAL command is used to produce a list of all externally loaded routines. In this case there is only one (SORT).
(18) The programmer is about to test the loaded external routine by using exactly the same immediate statements he used previously to test the internally compiled version (steps 8-10). Here he assigns new values to the immediate execution array A.
(19) Here the external routine is invoked with an immediate execution CALL statement.
(20) Control has returned from the external subroutine, and an expression statement is used to have the array A printed. The

external routine produced the same results as the internally compiled version.

(21) The /STOP command is entered, causing the IF run to be terminated, and control is returned to MTS.

MTS 6: FORTRAN in MTS

October 1983

October 1983

OVERDRIVE

INTRODUCTION

OVERDRIVE is a preprocessor which allows the use of structured programming techniques in FORTRAN 66 programs (i.e., *FTN and *IF).

FORTRAN was one of the first higher-level programming languages and, unfortunately, is today still handicapped by some of its earlier, archaic control structures. Structured programming has provided means to circumvent flow-of-control problems. Structured programming is concerned with representations of the source program which make the execution of the program easy to follow for the reader. It is characterized often as GOTOless programming using block- or grouping-type statements for condition testing and loops, thereby enabling one to see with less difficulty what path the execution flow follows.

The OVERDRIVE preprocessor extensions to FORTRAN use structured programming concepts to allow the clearer expression of a program's flow of control than is currently possible. Several listing control functions are also provided to help make the source program listings more readable.

OVERDRIVE may be conveniently used in conjunction with *FTN as described later.

Compatibility with FORTRAN 77

In March 1978 a new ANSI FORTRAN standard (referred to as FORTRAN 77) was approved to replace the 1966 FORTRAN standard (referred to as FORTRAN 66 or FORTRAN IV). OVERDRIVE allows the use of some of the FORTRAN 77 features and transforms the OVERDRIVE source into FORTRAN 66 source code. Where FORTRAN 66 and FORTRAN 77 differ in ways which are relevant to OVERDRIVE, these differences will be discussed.

The FORTRAN 77 features which can be specified in OVERDRIVE are the following:

(1) Block IF
(2) PARAMETER statement
(3) FMT= format specification
(4) Comment with *

These will be fully explained in following sections.

Additionally, OVERDRIVE contains many features not available in FORTRAN 77, primarily control structures and listing options. There are also many FORTRAN 77 features which were not feasible to add to OVERDRIVE.

Criteria for OVERDRIVE Features

The choice of OVERDRIVE extensions to FORTRAN was based on the following criteria for each extension:

(1) Useful to a large number of users
(2) Simple
(3) In the "style" of FORTRAN
(4) Easy to detect errors in
(5) Inexpensive to use
(6) Upward compatible with FORTRAN 66
(7) Compatible with FORTRAN 77

Portability

The problem of transporting OVERDRIVE programs to other installations can be addressed in two ways.

The output from a translation of an OVERDRIVE source program is FORTRAN 66-compatible and may be transported to other installations. OVERDRIVE may be directed to include source comments in its output so that the resulting program may be more easily read.

OVERDRIVE is written in SPITBOL and the source is available to those who wish to take it to another installation.

Definition of Terms

The following terms are used in the statement descriptions:

(1) ivar = unsubscripted integer variable
(2) icon = integer constant
(3) int = ivar | icon
(4) iexp = an integer valued expression
(5) lexp = a logical valued expression

October 1983

USAGE IN MTS

There are two ways in which OVERDRIVE may be used in MTS. It can be used either as an option of *FTN or run as a stand-alone program. With *FTN it is run with

```
$RUN *FTN SCARDS=source SPRINT=listing ...  PAR=OVER[=overopt]
```

where the MTS logical I/O unit assignments are the same as for a standard FORTRAN compilation. The OVER option in the PAR field indicates that OVERDRIVE preprocessing should be performed. Options (overopt) to be passed to OVERDRIVE may be specified in one of two forms. If there is just one option, it may be simply specified after an '=' following the OVER keyword; e.g., PAR=NOLIST,OVER=NOLIST (which suppresses both the FORTRAN and OVERDRIVE listings). If more than one option is to be specified, they must be in a comma-separated, parenthesized list; e.g., PAR=OVER=(COM,NOINDENT),OPT=2. Except for the options passed to OVERDRIVE in this manner, the remainder of the PAR field is ignored by OVERDRIVE.

As a stand-alone program, OVERDRIVE is run with $RUN *OVERDRIVE with the following MTS logical I/O unit assignments:

```
SCARDS - OVERDRIVE source
SPRINT - Listing output
     2 - Target module output in standard FORTRAN (to avoid conflicts with *FTN use of SPUNCH)
SERCOM - Error messages if running interactively
PAR=SIZE=30 - Allocates SPITBOL work space
```

SOURCE PROGRAM FORMAT

Input records must be in the following format:

```
Columns
  1      * or C indicating a comment
  1-5    Statement number
  6      Continuation flag
  7...   Statement

There is no limit on statement length.
No sequence-ID field is accepted.
```

Note that this corresponds to the *FTN option FORMAT=LONG. However, the FORMAT option in the PAR field passed to *FTN is ignored by OVERDRIVE. Any specification of the FORMAT option will not affect the way OVERDRIVE processes input records.

Additionally, no comment line may separate the portions of a continued line.

More than one module may be present in the OVERDRIVE input stream.

OVERDRIVE ignores the source program indentation and only looks at the statements to determine the scope of statements. Although the source program indentation is ignored, it may be easier to work with if indented properly.

SOURCE LISTING

OVERDRIVE produces a listing of the source program. This listing contains the FORTRAN Internal Statement Number (ISN), any generated label, and the MTS line number of each source line. The source lines are both indented and bracketed in the listing to show the scope of OVERDRIVE structures.

Internal Statement Numbers

OVERDRIVE prints the FORTRAN Internal Statement Number (ISN) along the left side of the listing. This Internal Statement Number will match either the FORTRAN G or FORTRAN H statement numbering scheme when running OVERDRIVE from *FTN. When running OVERDRIVE separately to produce output for FORTRAN H, it is necessary to specify the COMPILER=FTNH option (see the section "OPTION Statement").

File Line Numbers

The MTS file line number is printed on the left-hand side of the source listing.

Generated Labels

Generated labels are printed along the left-hand side of the source listing on the line which caused them to be defined. If there is more than one such label defined, the first will be printed and a '+' will be added to indicate that there were one or more unprinted generated labels. These labels are printed for those users who wish to use SDS with an OVERDRIVE program. One must be somewhat cautious in the use of these labels. If more than one target FORTRAN statement is generated by

an OVERDRIVE statement, the generated label which is printed may not necessarily be on the first of these. A cursory familiarity with the type of code generated by OVERDRIVE will suffice to make use of these labels with SDS.

Source Indentation

Quick recognition of the structure of the source program is essential to good programming. Each source line is indented to its proper structure level with a line of periods connecting the beginning and end of each structure. This scope bracket of periods may be changed by means of the INDENT keyword of the OPTION statement. Comments beginning with a 'C' or '*' in column 1 are not indented when the automatic indentation option is in use, however, the other forms of comments may be used to give uninterrupted scope brackets and indentation.

Continuation Lines

If the automatic indentation option is active (the default), the first line of the statement will be indented by the current indentation amount and each continuation line will be indented by the current indentation amount plus the amount that each was indented past the beginning of the first line of the statement.

Listing of FORMATs

For several reasons formats should not contain quoted fields which cross input line boundaries. In a system such as MTS, which truncates trailing blanks, it is impossible to see in the listing how many blanks should be in the quoted field at the point of continuation. In OVERDRIVE the automatic indentation feature may in this case make it appear in the listing as though extra characters are inserted at that point in the format, although the target module will contain the original input number of characters.

TARGET MODULE

When run from *FTN, OVERDRIVE always generates the target module in the temporary file -OVEROBJ after first emptying it. See the section "MTS Usage" for a description of the stand-alone version.

The target module file line numbers are taken, as nearly as possible, from the source file line numbers. These numbers may differ slightly in the area of continued and inserted statements.

Comments in the source program are carried over to the target program only if the COM option (see OPTION statement) has been selected. Indentation in the target code matches that of the source code.

Created Labels

In translating OVERDRIVE statements into standard FORTRAN it is necessary to create statement labels. One of two methods may be selected for the creation of these labels.

(1) The default method (OPTION LABEL=LINE) generates each statement label starting with a prefix digit (default '9') followed by a suffix which is based on the source file line number. For example, a label generated at line 10 would be 910. This suffix is either the integer portion of the source file line number or, if that has already been generated or is nonpositive, one higher than the last generated label. This allows ranges of fractional line numbers as well as concatenations of files. This scheme has the dual advantages of relating the labels to source file line numbers for mnemonic convenience and preventing recompilations with minor source changes from changing all generated labels. Therefore, it is usually easier to use this method when debugging with SDS or *IF.

    If the line number is greater than 89999, then OVERDRIVE will switch to the alternate method of label generation, counting down from 99999 for all remaining created labels.

    Note that these labels will use the file line number where they are first referenced, not necessarily where they are defined in the label field. For example, the IF statement would create a label based on the current source line number but the label would not be defined until the next corresponding ELSEIF, ELSE, or ENDIF statement.

    The label prefix may be changed (default '9') by means of the OPTION LPFX=x statement where "x" is a digit in the range 1 to 9.

(2) The alternate method generates labels from 99999 in decreasing sequence. The default initial value of 99999 may be changed with the OPTION LABEL=icon statement where the initial value to count down from is specified by icon.

    Using this alternate method of label generation, the label counter is not reset between compilations. This avoids conflicting labels when debugging with SDS.

Regardless of which method is used, it is inadvisable for the user to define labels beginning with a '9'. Any source program definition of a label which is identical to one generated by OVERDRIVE will result in a FORTRAN error. Any source program usage, without definition, of a label generated by OVERDRIVE may not result in a error message from either OVERDRIVE or FORTRAN, but will very likely cause an error in execution.

Created Integer Variables

Integer variables are created by using the prefix 'I' followed by a number which is initially 99999 and is counted down by one for each new variable. Because these variable names are created at the time they are needed and since OVERDRIVE makes only one pass over the source program, no declarations can be made at the beginning of the program. Therefore the implicit type of variables beginning with 'I' must not be changed.

Target Module Code

Prototypes of the code generated by OVERDRIVE are given in the descriptions of many of the OVERDRIVE statements. The precise code generated may be slightly different in form but will perform the equivalent action.

CONTROL STRUCTURES

Four types of control structures are supplied in OVERDRIVE:

(1) An IF structure for selecting groups of statements depending on one or more logical expressions.
(2) A CASE structure which allows one of many sections of statements to be selected by number.
(3) A LOOP structure which provides a means of repeating sections of code.
(4) A PROCEDURE structure which allows use of a common section of code in several places.

IF...ENDIF

Augmenting the FORTRAN logical and arithmetic IF statements are the block IF structures. These IF structures are compatible with those of

FORTRAN 77 with the exception that FORTRAN 77 does not allow GOTOs into the scope of an IF structure. OVERDRIVE has no such restriction. Indeed, in order to allow internal procedures, it is necessary to allow such GOTOs.

The simple IF structure is used when the execution of a section of code depends on the value of a logical expression.

```
          IF (lexp) THEN
             statements to be executed if lexp is true.
          ENDIF
```

This is translated into

```
          IF (.NOT.(lexp)) GO TO xxxxx
             statements to be executed if lexp is true.
    xxxxx CONTINUE
```

Example:

```
          IF (I.LT.N) THEN
             A(I) = VAL
             I = I+1
          ENDIF
```

IF...ELSE...ENDIF

The IF...ELSE...ENDIF structure is used when one of two sections of code are to be selected based on the value of one condition. The ELSE statement may not be labeled.

```
          IF (lexp) THEN
             statements to be executed if lexp is true
          ELSE
             statements to be executed if lexp is false
          ENDIF
```

This is translated into:

```
          IF (.NOT.(lexp)) GO TO xxxxx
             statements to be executed if lexp is true
             GO TO yyyyy
    xxxxx CONTINUE
             statements to be executed if lexp is false
    yyyyy CONTINUE
```

Example:

```
          IF (LEVEL.GT.1) THEN
             REFUTE = MAXVAL-REF(LEVEL)
             DOREF = .TRUE.
          ELSE
```

```
            REFUTE = -1
            DOREF = .FALSE.
         ENDIF
```

IF...ELSEIF...ENDIF

An IF...ELSEIF...ENDIF structure may be used when selecting among many parallel sections of code with many parallel conditions. This language construction can be equivalently coded with IF statements within the ELSE clauses of the preceding IF. However, such testing of parallel conditions is a common program flow structure that should be expressed in an easily seen manner as with the IF...ELSEIF...ENDIF construction.

The conditions are tested sequentially until one is found to be true.

The ELSEIF statement may not be labeled. An optional ELSE clause may be used to specify statements to be executed if none of the preceding conditions were true.

```
         IF (lexp1) THEN
            statements to be executed if lexp1 is true
         ELSEIF (lexp2) THEN
            statements to be executed if lexp1 is false
            and lexp2 is true
         ELSEIF (lexp3) THEN
            statements to be executed if lexp1 and lexp2
            are false and lexp3 is true
         ...
         as many ELSEIF statements as necessary
         ...
         [ELSE]
            statements to be executed if none of the
            preceding lexps was true
         ENDIF
```

This translates into the following:

```
         IF (.NOT.(lexp1)) GO TO xxxxx
            statements to be executed if lexp1 is true
            GO TO aaaaa
   xxxxx IF (.NOT.(lexp2)) GO TO yyyyy
            statements to be executed if lexp1 is false
            and lexp2 is true
            GO TO aaaaa
   yyyyy IF (.NOT.(lexp)) GO TO zzzzz
            statements to be executed if lexp1 and lexp2
            are false and lexp3 is true
            GO TO aaaaa
            ...
   [qqqqq CONTINUE]
            statements to be executed if none of the
```

```
            preceding lexps is true
     aaaaa CONTINUE
```

Example:

```
           IF (STMTYP.EQ.COMMNT) THEN
              CALL COPSTM
           ELSEIF (STMTYP.EQ.CONTINU) THEN
              CALL IGNORE
           ELSE
              CALL REGSTM
           ENDIF
```

DOCASE...ENDCASE

The DOCASE statement permits the selection of one of a number of groups of statements depending upon the value of an integer expression given in the DOCASE statement. Control passes to the particular CASE statement specifying that control value and then, unless otherwise exited, continues execution following the ENDCASE statement. If the control value is not specified in any CASE statement, execution proceeds with the ELSECASE clause. If there is no ELSECASE, execution continues past the ENDCASE. Because no error will be reported if the control value is not covered by any case and because this situation is often an error, it is usually a good idea to provide an ELSECASE clause to give some meaningful error message.

Unlike a series of ELSEIF constructions, the DOCASE statement does not test for each case sequentially, but instead uses a computed GOTO to go directly to the correct case.

Do not use the DOCASE statement when the case values are sparsely spread over a large region. Because one label is put onto a computed GOTO for each possible control value, an impractically large statement would be generated in this situation.

The DOCASE construction must always begin with a DOCASE statement. Next are one or more CASE statements, each preceding a group of statements to be executed if the control value is specified on that CASE. The ELSECASE statement is next, if present, and finally, the ENDCASE statement, terminating the DOCASE...ENDCASE structure.

No label may occur on either the CASE or ELSECASE statements.

```
      DOCASE (iexp)
      CASE (icon,...)
         statements to be executed if iexp
         equals any of the icons
      CASE (icon,...)
         statements to be executed if iexp
```

```
         equals any of the icons
      ...
      as many cases as required
      ...
      [ELSECASE]
         If present, the statements following are executed if
            (a)  iexp < min(icons) or
            (b)  iexp > max(icons) or
            (c)  iexp = unspecified icon.
         If there is no ELSECASE statement, execution will proceed
         following the ENDCASE statement under these conditions.
      ENDCASE
```

Execution of the DOCASE causes a branch to prolog code which is generated at the ELSECASE statement. If there is no ELSECASE statement, the prolog code is generated at the ENDCASE statement. After checking the range of the DOCASE control value, the appropriate case is branched to.

Example:

```
          DOCASE (SYS)
          CASE (1)
             *** SYSTEM IS VMOS ***
             EVTIME = .140
             INPUT  = 1
             OUTPUT = 2
          CASE (2,5)
             *** SYSTEM IS MTS ***
             EVTIME = .0085
             INPUT  = 5
             OUTPUT = 6
             CALL DEFPRT(0)
          ELSECASE
             *** UNKNOWN SYSTEM ***
             EVTIME = 0.
             INPUT  = 5
             OUTPUT = 6
          ENDCASE
```

Generated code from above example

```
          GO TO 91
     92   EVTIME = .140
          INPUT  = 1
          OUTPUT = 2
          GO TO 97
     98   EVTIME = .0085
          INPUT  = 5
          OUTPUT = 6
          CALL DEFPRT(0)
          GO TO 97
     91   IF(SYS.GE.1.AND.SYS.LE.5)GOTO(92,98,913,913,98),SYS
```

```
913   EVTIME = 0.
      INPUT  = 5
      OUTPUT = 6
97    CONTINUE
```

Loop Structures

The only looping statement in FORTRAN 66, as well as FORTRAN 77, is the iterated DO, which unfortunately requires a statement label to mark its termination. OVERDRIVE offers a similar iteration statement in a labelless form (LOOP), as well as means of testing conditions at the beginning and/or end of the loop and exiting while setting a flag.

All looping is performed using the LOOP...ENDLOOP structure. Clauses may be specified on the LOOP and ENDLOOP statements which control the continuation or termination of the execution of the loop.

Extended ranges of DO loops (i.e., branches out-of and then back into an active DO loop) are prohibited in FORTRAN 77 and have limitations in FORTRAN 66. OVERDRIVE makes no attempt to enforce this restriction and, indeed, would not do so because it would then not be possible to make internal procedure calls. The target code generated for loops by OVERDRIVE puts no restriction on actions performed in internal procedures called from within an OVERDRIVE loop; that is, the extended range restriction of FORTRAN 66 does not apply.

The legal clauses are:

```
LOOP [for] [while] [until] [exit]
ENDLOOP [REPEAT [while] [until]]
```

Clauses specified on the LOOP statement are tested at the beginning of each iteration of the loop. Clauses on the ENDLOOP statement are tested at the end of each loop iteration. If more than one LOOP or ENDLOOP clause is given, the looping continues only if all clauses would continue execution.

LOOP

The unembellished LOOP statement makes no tests at the top of the loop. It is possible to specify conditions on the ENDLOOP to be tested at the bottom of the loop. Without termination conditions on either the LOOP or ENDLOOP statements, infinite looping must be avoided by a statement within the loop which branches out, such as: EXITLOOP, GOTO, RETURN, or STOP. Of these, EXITLOOP would be preferable from a structured programming point of view.

Example:

```
LOOP
   *** COUNT TO INFINITY ***
   I = I + 1
ENDLOOP
```

LOOP FOR(iteration)

The LOOP...ENDLOOP structure with a for-clause is a labelless replacement for the FORTRAN DO statement. It removes several irritating restrictions of the FORTRAN 66 DO loop (e.g., expressions are allowed). It is in general conformance with execution of the FORTRAN 77 DO loop, which differs in several respects from the FORTRAN 66 DO loop. Specifically, the FORTRAN 77 loop termination test is made before the first iteration is made and the iteration variable is defined upon termination. Unlike FORTRAN 77, OVERDRIVE allows only integer iteration values.

```
LOOP FOR (ivar=iexp1,iexp2[,iexp3])
   statements in the body of the loop
ENDLOOP
```

where iexp1 is the initial value, iexp2 is the maximum value, and iexp3 is the increment (default 1).

The exact translation depends on the nature of iexp2 and iexp3. Changes to either of these values during the execution of the loop is defined in FORTRAN 77 to have no effect on the number of times the loop is executed. To conform with this, if either iexp2 or iexp3 references a variable, copies of the initial values of these expressions are made and then used in their places in the following prototype.

Regardless of their order in the source statement, the FOR and EXIT clauses are processed before any others as these are the only LOOP clauses which require initialization. It is because of this initialization that the FOR and EXIT clauses are banned from the ENDLOOP.

```
      ivar = iexp1
xxxxx IF (ivar.gt.iexp2) GO TO yyyyy
         statements in the body of the loop
      ivar = ivar + iexp3
      GOTO xxxxx
yyyyy CONTINUE
```

LOOP WHILE(lexp)

The LOOP WHILE (lexp) statement is for specifying a loop which is continued as long as the condition lexp is true at the beginning of each iteration.

```
          LOOP WHILE (lexp)
             statements to be executed within loop
          ENDLOOP
```

is translated to:

```
     xxxxx IF(.NOT.(lexp)) GO TO yyyyy
             statements to be executed within loop
           GOTO xxxxx
     yyyyy CONTINUE
```

LOOP UNTIL(lexp)

The LOOP UNTIL (lexp) statement is for specifying a loop which is terminated at the beginning of any iteration in which the condition lexp is true.

```
          LOOP UNTIL (lexp)
            statements to be executed within loop
          ENDLOOP
```

is translated to:

```
     xxxxx IF (lexp) GO TO yyyyy
             statements to be executed within loop
           GOTO xxxxx
     yyyyy CONTINUE
```

LOOP EXIT(signal,...)

The EXIT clause on the LOOP statement serves two functions. First, it provides a means of identifying the loop. This identification can then be used by the EXITLOOP or NEXTLOOP statements to identify which of several imbedded loops the action applies to. Second, it provides a means of recording a special condition which caused the loop termination.

A list of signals is specified in the LOOP EXIT clause statement. Each signal may be either a logical variable or a character string (e.g., 'MOOD INDIGO').

Either form of signal (logical variable or character string) may be used to identify a loop for the EXITLOOP or NEXTLOOP statements. A logical variable can additionally be used to record that a specific condition caused a loop to be exited.

Each logical variable specified in the EXIT clause is set to false at the beginning of the loop. The execution of an EXITLOOP statement specifying one of these logical variables will cause the variable to be set to true and the loop to be exited. The NEXTLOOP statement does not alter the value of any of these logical variables.

Logical exit variables are not tested as one of the conditions for proceeding with the next LOOP iteration. They are set false at the beginning of the loop and may be set true by the EXITLOOP statement, but never tested as part of the LOOP code.

It is the responsibility of the user to declare exit variables to be of type LOGICAL.

Example: Typical Table Searching

```
LOOP FOR(I=1,N) EXIT(FOUND)
   *** SEARCH UNTIL A MATCH IS FOUND ***
   IF (T(I).EQ.ITEM) THEN
      EXITLOOP (FOUND)
   ENDIF
ENDLOOP
IF (.NOT.FOUND) THEN
   *** ADD TO END OF TABLE ***
   T(I) = ITEM
   N = N + 1
ENDIF
```

Example:

```
LOOP FOR (I=1,N) EXIT ('QQSV')
   LOOP FOR (J=1,M)
      ...
      IF (A(I,J).EQ.-1) THEN
         EXITLOOP ('QQSV')
      ENDIF
      ...
   ENDLOOP
ENDLOOP
```

ENDLOOP [REPEAT [while] [until]]

An ENDLOOP statement must be supplied to match each LOOP statement. The ENDLOOP statement may be written without any termination conditions, in which case it will generate a branch back to the top of the loop.

A termination condition may be specified with either a WHILE or UNTIL clause on the ENDLOOP. In this case, the word REPEAT must be inserted after the ENDLOOP keyword and before these clauses.

ENDLOOP REPEAT WHILE(lexp) tests the condition lexp at the end of each iteration and continues with the next iteration of the loop only if the condition lexp is true.

ENDLOOP REPEAT UNTIL(lexp) tests the condition lexp at the end of each iteration and terminates execution of the loop only if the condition lexp is true.

Both WHILE and UNTIL clauses may be specified on the ENDLOOP statement. The loop will be continued only if the WHILE lexp is true and the UNTIL lexp is false.

The REPEAT keyword is required to prevent misinterpreting ENDLOOP WHILE(lexp) as meaning terminate the loop when lexp is true. ENDLOOP REPEAT WHILE(lexp) does not suffer from such a problem in interpretation.

EXITLOOP [(signal)]

The EXITLOOP statement causes execution to continue with the statement immediately following the end of an enclosing LOOP. The signal, if present, must be specified in the EXIT clause of an enclosing LOOP.

If no signal is specified on the EXITLOOP statement, the innermost enclosing LOOP will be exited.

If the signal is a logical variable, that variable will be set to true and the enclosing loop which specified that variable in its EXIT clause will be exited.

If the signal is a character string, execution will continue after the ENDLOOP of the enclosing loop which specifies the signal in its EXIT clause.

Example:

```
LOOP FOR(I=1,N) EXIT(X)
   LOOP FOR(J=1,M)
      ...
      EXITLOOP (X)
      ...
   ENDLOOP
ENDLOOP
```

If the EXITLOOP statement in this example is executed, the outermost loop will be exited with X set to true. If the EXITLOOP statement is never executed and the loops are terminated by exhausting the iterations, the value of X will be false.

NEXTLOOP [(signal)]

The NEXTLOOP statement causes execution to continue with the next iteration of the enclosing loop. All LOOP and ENDLOOP conditions will be tested before the next iteration is performed. The signal, if present, must be specified in the EXIT clause of an enclosing LOOP.

If no signal is specified on the NEXTLOOP statement, the next iteration of the innermost enclosing LOOP will be started.

The signal may be either a logical variable or a character string. The signal is used simply to identify the loop and, if a variable, will not be changed.

Example:

```
LOOP FOR(I=1,N) EXIT('NEXTI')
   LOOP FOR(J=1,M)
      ...
      NEXTLOOP ('NEXTI')
      ...
   ENDLOOP
ENDLOOP
```

If the NEXTLOOP statement in this example is executed, the next iteration of the outermost loop will be started without waiting for the inner loop to be terminated by going through all values of J.

Internal Procedures

This facility provides for procedure definition and linkage within a FORTRAN main or subprogram.

Definition of procedures may occur anywhere an executable statement may appear except within an OVERDRIVE structure. Placement of a procedure definition in the execution flow path is inadvisable because it will make reading the program difficult. However, since a branch will be generated around the procedure definition in this case, execution will not be disrupted by falling into a procedure body.

Parameters are not allowed on an internal procedure. However, since all variables are global to a procedure, parameters may be passed by assigning them to variables. An internal procedure also produces no result as a function would. The results of a procedure must be passed back in variables.

Invocation (calling) of a procedure may be made either explicitly with the INVOKE statement or implicitly by simply specifying the procedure name if that name contains underscore characters.

Recursive procedure invocation is not allowed and, if attempted, may result in an infinite execution loop. There is no limit on procedure calling depth.

Listing of each procedure is set off from the preceding and following text with a few blank lines and a dashed line. A procedure cross-reference is produced at the end of the listing.

PROCEDURE...ENDPROCEDURE

The body of an internal procedure definition is enclosed within a PROCEDURE...ENDPROCEDURE structure.

```
PROCEDURE pname
   the body of the procedure
ENDPROCEDURE
```

The procedure name pname must begin with an alphabetic character and may be continued with alphabetics, digits, or underscores to a maximum length of 32 characters.

PROC is a valid abbreviation for PROCEDURE, ENDPROC for ENDPROCEDURE, and EXITPROC for EXITPROCEDURE.

Execution of the ENDPROCEDURE statement causes execution to return to the point of the procedure call. The EXITPROCEDURE statement may be used to exit a procedure before reaching the ENDPROCEDURE statement.

The generated FORTRAN code is:

```
xxxxx CONTINUE
         the body of the procedure
         GO TO zzzzz  goes to the epilog at program end
         ...
zzzzz GO TO Ixxxxx,(list of return labels)    returns to caller
```

Example:

```
PROCEDURE SUM_ARRAY
*** COMPUTE OF ALL ELEMENTS IN THE ARRAY A ***
SUM = 0
LOOP FOR (I=1,N)
   SUM = SUM + A(I)
ENDLOOP
  ENDPROCEDURE
```

Calling an Internal Procedure

There are two methods of calling an internal procedure. It is possible to write an explicit call using the INVOKE statement or to write an implicit call by simply writing the procedure name. The implicit form of the call is valid only if the procedure name contains one or more underscore characters.

Form of an explicit call:

```
INVOKE pname
```

Form of an implicit call:

pname

Both explicit and implicit calls translate into:

```
        ASSIGN yyyyy TO Izzzzz
        GO TO xxxxx     go to the procedure
  yyyyy CONTINUE        return is to here
```

Example:

```
        INVOKE SUM_ARRAY or
        SUM_ARRAY
```

EXITPROCEDURE statement

The EXITPROCEDURE statement provides a means of exiting an internal procedure without executing the ENDPROCEDURE statement. EXITPROC is a valid abbreviation for EXITPROCEDURE.

Example:

```
      PROCEDURE A
        LOOP WHILE (X.NE.Y)
           ...
           IF (X.EQ.0) THEN
              EXITPROCEDURE
           ENDIF
        ENDLOOP
        ...
      ENDPROCEDURE
```

FORMATS

When labelless control statements for the common flow patterns are used, statement labels become important in signaling the presence of some unusual flow. Since FORMAT labels are often distracting when trying to follow program flow, OVERDRIVE has implemented the FMT= feature of FORTRAN 77. OVERDRIVE extends the facility for multiple occurrences of the same format in close proximity.

An additional facility, similar to FMT=, for specifying formats imbedded in READ and WRITE statements is also provided.

FMT= Format Specification

In OVERDRIVE, as in FORTRAN 77, formats may be imbedded within READ and WRITE statements. This is done by placing, in the position that the format number would normally occupy, FMT= followed by the format as a quoted character constant. Quoted character constants are enclosed in single quotes and all such quotes occurring in the character constant must be represented by a pair of quotes. OVERDRIVE requires these formats to contain no counted Hollerith (H) fields.

Example:

```
          WRITE (OUTPUT,22) I
    22    FORMAT (' *** ',I3,' BLOCKS')
```

could be equivalently written as

```
          WRITE (OUTPUT,FMT='' *** '',I3,'' BLOCKS'') I
```

Imbedded Formats

In OVERDRIVE formats may be imbedded in the READ and WRITE statements at the point where the FORMAT number would otherwise occur. The format must be enclosed in parentheses and may not contain any counted H fields.

Example:

```
          WRITE (OUTPUT,22) I
    22    FORMAT (' *** ',I3,' BLOCKS')
```

could be equivalently written as

```
          WRITE (OUTPUT,(' *** ',I3,' BLOCKS')) I
```

Implied Formats

If a format is to be used more than once in a local section of code, it is possible to write the format only one time and have that format use implied in subsequent READ and WRITE statements. In this way, the confusing appearance of a label in the middle of executable statements is avoided. It also obviates the practice of putting all of the formats at the end or beginning of a program to get their labels out of the execution path. An implied format is defined in one of three ways:

(1) As the format specified by FMT= in a READ or WRITE statement.
(2) As an imbedded format in a READ or WRITE statement.
(3) As a FORMAT statement appearing without any statement number.

The definition of an implied format remains active until either redefined by another format, or until it expires after 10 statements. Restriction of implied format definitions to the locality of the last 10 statements prevents both inadvertent errors and difficult-to-read programs.

Once an implied format is defined, it will be used in any READ or WRITE statement in which a necessary format or format number has been omitted.

Examples:

```
          WRITE (OUTPUT,22) I
          WRITE (PRINTR,22) I
     22   FORMAT (' ***',I3,' BLOCKS ALLOCATED')
```

could be written in any of the following ways:

```
          WRITE (OUTPUT,FMT='' ***'',I3,'' BLOCKS ALLOCATED'')' I
          WRITE (PRINTR,) I
```

or

```
          WRITE (OUTPUT,(' ***',I3,' BLOCKS ALLOCATED')) I
          WRITE (PRINTR,) I
```

or

```
          FORMAT (' ***',I3,' BLOCKS ALLOCATED')
          WRITE (OUTPUT,) I
          WRITE (PRINTR,) I
```

PARAMETER STATEMENT

The PARAMETER statement is used to give a constant a symbolic name. The form is:

```
     PARAMETER (p=c [,p=c]...)
```

where p is a symbolic name, one to six characters beginning with an alphabetic and continuing with alphanumerics.
c is a constant.

Each p is the symbolic name associated with a constant c. Each subsequent occurrence of p in the source program text will be replaced by the constant c. The listing will be produced showing the symbolic name.

Each c may be one of the following constants: integer, single- or double-precision real, complex, logical, or character. Of these constants only character constants may contain imbedded blanks. Character constants are enclosed within single-quote characters and may not be written in the counted Hollerith form.

As in FORTRAN 77, parameter substitution will not take place in formats.

The OVERDRIVE PARAMETER statement is very similar to that in FORTRAN 77. There are, however, several differences. OVERDRIVE restricts the right-hand side to be a constant (FORTRAN 77 allows a constant expression). FORTRAN 77 requires that the type of the parameter name be agreeable with the type of the constant expression, either by explicitly declaring the name or by implicit default. OVERDRIVE makes no type-compatibility check.

It is the intention that, when a FORTRAN 77 compiler becomes available, OVERDRIVE will cease to process the PARAMETER statement and leave that job to the FORTRAN compiler. This means that the user should adhere to the FORTRAN 77 rules regarding type when using the PARAMETER statement.

Warning: OVERDRIVE, unlike FORTRAN 77, does the parameter substitution before parsing each statement. This means that parameter names must not be the same as any FORTRAN or OVERDRIVE keywords. Since blank elimination is not done by OVERDRIVE, blanks appearing in the middle of variable names or keywords may cause a false parameter match.

The PARAMETER statement must not have a label.

Example:

```
      PARAMETER (BSIZE=250)
      INTEGER B(BSIZE)
```

would generate

```
      INTEGER B(250)
```

COMMENT STATEMENTS

In addition to the regular FORTRAN comments two additional types of comment statements have been provided in OVERDRIVE. These additional comments have three advantages: (1) They do not obscure program flow by cluttering the label field, (2) they do not interrupt the scope brackets which are printed out when using the automatic indentation feature, and (3) they are indented to the current indentation level.

The COM and NOCOM options (see the OPTION statement) are used to control the passage of source program comments into the FORTRAN target program. The default value is NOCOM which inhibits the passage of any comments into the target program. If COM is specified, comments will be passed on into the target program, replacing the first character with a 'C' if necessary. Comment lines may never be continued.

FORTRAN Comment Lines

Any line beginning with either a 'C' or an '*' (FORTRAN 77 allows an '*') in column 1 is considered to be a FORTRAN comment and is listed by OVERDRIVE without indentation or scope brackets.

Blank lines are also considered to be comments in FORTRAN 77 and are treated as such by OVERDRIVE. Unlike the other FORTRAN comments, blank lines do not interrupt the listing of scope brackets.

OVERDRIVE Comment Lines

Any statement that begins with an asterisk ('*') in column 7 or beyond is treated as a comment by OVERDRIVE. The advantage of these comment lines over those which begin with a 'C' or '*' in column 1 is that these comments are indented to the current structure nesting depth and they do not interrupt the scope brackets. Use of these comments will, in general, produce a more readable listing.

Note that these comments must begin in column 7 or beyond. An '*' in columns 2 through 5 will be treated as an error.

OVERDRIVE Appended Comments

A comment may be appended to the end of any FORTRAN or OVERDRIVE statement by separating it from the statement with the two characters ';*'. Everything following the ';*' will be treated as a comment.

LISTING CONTROL STATEMENTS

The spirit of easily readable programs is well served with some means of formatting the program source listing. To that end, the following statements are interpreted by OVERDRIVE. None of these generates any executable code.

None of the listing control statements may be labeled.

EJECT [icon]

The EJECT statement without a parameter causes the listing to continue at the top of the next page, if not already at the top of a page. The current title and subtitle, if any, will be printed at the top. The occurrence of a new subprogram or BLOCK DATA also causes the EJECT action.

When EJECT has an integer parameter, the EJECT action is only taken if there are fewer than icon lines remaining on the current page. This is useful where there is a section of code or comments which should not be broken across a page boundary but which need not start a new page.

TITLE 'text of title'

The TITLE statement makes the text contained between the quotes into the current title and then causes the same action as the EJECT statement.

The subtitle text is blanked by the occurrence of a TITLE statement.

If no TITLE is issued, the name of the subprogram will be used as the default title text.

SUBTITLE 'text of subtitle'

The SUBTITLE statement makes the text contained between the quotes into the current subtitle and then causes the same action as the EJECT statement. The EJECT action is not taken if the listing is already positioned at the top of the page as would be the case if a TITLE statement appeared immediately previous to this.

SPACE icon

The SPACE statement causes icon number of blank lines (or an EJECT to the top of the next page if there are fewer than icon lines remaining on the current page) to be generated at this point in the source listing.

An all-blank line is to be preferred to a SPACE 1 because, although both produce a blank line in the OVERDRIVE listing, a blank line makes the text in the source file more readable.

Blank lines will not be printed at the top of a page.

LIST [option]

The LIST statement may be used to turn the source listing switch on or off. When the listing switch is on, a source listing will be printed; when off, it will not be printed. This listing switch is initially on. The following options may be specified:

ON	the listing switch is turned on.
OFF	the listing switch is turned off.
PUSHON	the listing switch is saved on a stack and turned on.
PUSHOFF	the listing switch is saved on a stack and turned off.
POP	the listing switch is restored from the stack.

If the option is omitted, ON is defaulted.

INDENT...ENDINDENT

The INDENT...ENDINDENT structure causes the enclosed statements to be listed one indentation level deeper than they otherwise would be. This structure only affects the listing and does not produce any code in the target module.

OPTION STATEMENT

The OPTION keyword is followed by a comma-separated list of the options to be selected.

Examples:

```
OPTION COM,NOXREF
OPTION INDENT='|   '
```

OPTION {COM|NOCOM}

Default: NOCOM

This option controls the passage of comments into the target module. If COM is specified, all comments in the source module are passed into the target module. If NOCOM is specified, no comments are passed to the target module.

Internal procedure calls and definitions will also generate comments in the target module if the COM option is in effect.

OPTION COMPILER={FTNG|FTNH}

Default: COMPILER=FTNG

The COMPILER option allows the user to specify which compiler the listing and target module should be produced for. Currently, this only affects the Internal Statement Number (ISN) computation which differs between FORTRAN G and FORTRAN H. This option is passed to OVERDRIVE by *FTN so it is only necessary to specify this option when running *OVERDRIVE alone and when producing a listing for FORTRAN H.

OPTION {INDENT=string|NOINDENT}

Default: INDENT='. '

The INDENT option is followed by an '=' and a quoted string. This string is taken as the string to be replicated once for each indentation level. Automatic indentation may be turned off with the NOINDENT option.

OPTION LABEL={LINE|icon}

Default: LABEL=LINE

The generation of labels is based either on the line numbers of the source file or on an arbitrary counting scheme. The default method uses the source file line numbers. The alternate method counts down from a number specified by the user with the LABEL=icon option. Probably the only reason for using the LABEL=icon option is to insure that generated labels are concentrated in one range and will therefore not conflict with user labels.

A complete explanation of the label-generation algorithm can be found in the section entitled "Created Labels."

Example:

```
OPTION LABEL=99999
```

OPTION {LIST|NOLIST}

Default: LIST

This option controls all further source program listing, either suppressing it with NOLIST or enabling it with LIST. It performs the same action as the LIST {ON|OFF} statement.

OPTION LPFX=digit

Default: LPFX=9

This option can be used to change the prefix digit that is put at the beginning of labels created from source file line numbers. This may be done to prevent conflicts with user labels. It also may be set to different values for different modules so there is no label conflict when using SDS.

A complete explanation of the label generation algorithm can be found in the section entitled "Created Labels."

OPTION {XREF|NOXREF}

Default: XREF

The XREF option produces a cross-reference of internal procedure definitions and invocations at the end of the source program listing.

EFFICIENCY CONSIDERATIONS

In the interest of speed, OVERDRIVE performs a one-pass translation. Because of the nature of the control structures being translated, some minor inefficiencies may be generated in the OVERDRIVE target module. These inefficiencies are so minor as to be of no consequence for all but the most unusual program. However, they are described below.

Extra GOTOs

An extra GOTO is always executed in a CASE statement because the labels in the computed GOTO are not all known until the end of the case structure. Similarly, an extra GOTO must be generated for the ENDPROCEDURE and EXITPROCEDURE statements because the return labels for the assigned GOTO which does the return are not known until the end of the program. The NEXTLOOP statement may also generate an extra GOTO if there is no condition specified on the ENDLOOP statement.

Extra Integer Temporaries

Each time a temporary variable for internal procedure return labels and LOOP and DOCASE temporaries for expressions is needed, it might, in some instances, be possible to reuse previously created variables. It is, however, not possible for OVERDRIVE to economically determine when such reuse would be safe and a new one is created for every new occasion. If a user should decide by inspecting the target module that there are too many redundant integer temporaries, he may recode the LOOP and DOCASE statements substituting his own temporary variables in place of the integer expressions.

RESTRICTIONS

The following restrictions apply to OVERDRIVE source programs.

Statement Numbers

User-defined labels should not conflict with labels created by OVERDRIVE. See the section on "Created Labels" for a discussion of ways to avoid conflicts.

Integer Variables

The source program must not contain any references to integer variables of the form generated by OVERDRIVE. These begin with an 'I' and are followed by a five-digit number starting with 99999 and counting down.

Because both the FORTRAN 66 and FORTRAN 77 standards allow declarations only at the beginning of the program, OVERDRIVE is unable to generate declarations for these created integer variables. Instead, it relies on the implicit INTEGER typing of variables beginning with 'I'. The user should not alter, with the IMPLICIT statement, the type of variables beginning with 'I'.

Significant Blanks

Unlike FORTRAN, blanks are used by OVERDRIVE to separate OVERDRIVE keywords from other alphabetic text, e.g., LOOP WHILE(...) must have one or more blanks between the words LOOP and WHILE. An exception to this is made for the OVERDRIVE keywords beginning with END; e.g., ENDLOOP. It is permitted to separate the END from the following portion of the keyword by a blank, e.g., ENDIF and END IF are equivalent.

Blanks are not permitted in logical operators, parameter names, or procedure names. Blanks used in the middle of variable names will cause erroneous processing if any blank-separated part of the variable name is the same as a PARAMETER name.

Comments Intermixed with Continuation Lines

Comment lines may not be intermixed with continuation lines; i.e., no comment line of any form may immediately precede a continuation line. This restriction applies to all OVERDRIVE and FORTRAN statements. If one were to use such a construction, the translation would result in either an OVERDRIVE or a FORTRAN error message.

Reserved Words

No OVERDRIVE keyword should be used as a variable in the source program. Such use could result in an erroneous translation.

Labeled OVERDRIVE Statements

Statement labels may not occur on the following OVERDRIVE statements:

Listing control statements

```
EJECT
SPACE
TITLE
SUBTITLE
INDENT
ENDINDENT
LIST
```

Statements with hidden branches

```
CASE
ELSE
ELSECASE
ELSEIF
```

Declarative statements

OPTION
PARAMETER
PROCEDURE

A label would not be very meaningful on any of the listing control statements or declarative statements. A label is confusing on each of the other statements because a GOTO is the first part of the generated code.

Counted Hollerith Values

No OVERDRIVE statement may contain counted Hollerith (H) values which contain either a quote or a parenthesis.

Error Handling

Error detection by the preprocessor is not performed on those portions of the source text which are presumed to be FORTRAN. This can result in FORTRAN errors on what were intended to be OVERDRIVE statements. This can happen because of a minor spelling or punctuation error in the OVERDRIVE statement which was simply passed on to FORTRAN with no processing by OVERDRIVE.

Errors in an OVERDRIVE structure may result in further errors as a result of structure mismatch propagation.

When used with *FTN, any serious error detected by OVERDRIVE inhibits further processing by FORTRAN, although OVERDRIVE will complete the listing and translation of the source. Minor errors, such as an error in the TITLE statement, will not inhibit further FORTRAN processing.

Error messages are printed in the listing both at the point of detection and at the end. If OVERDRIVE is being run in interactive mode, errors will also be listed on SERCOM.

October 1983

APPENDIX A: EXAMPLE OVERDRIVE PROGRAM

On the following pages is a portion of an indented OVERDRIVE listing. It does not include the line and ISN numbers nor the page header.

```
 IF (LEVEL.LT.VPLYMN) THEN
 .   *** DO EXCHANGE SORT ***
 .   CALL ORDER2(NEWLOW,UPPER,UPPER-NEWLOW)
 .   IF (NEWLOW.NE.LOWER) CALL MERGE(LOWER,NEWLOW,UPPER)
 ELSE
 .   IF (LEVEL.NE.VPLYMX) THEN
 .   .   CALL ORDER2(NEWLOW,UPPER,UPPER-NEWLOW)
 .   .   IF (VALUEF(NEWLOW).GE.0 .AND. VALUEF(UPPER).LE.0) THEN
 .   .   .   *** MIXTURE OF TURBS AND QUIETS ***
 .   .   .   LOOP FOR (I=NEWLOW,UPPER) EXIT(P)
 .   .   .   .   *** FIND FIRST TURBULENT ***
 .   .   .   .   IF (VALUEF(I).LE.0) THEN
 .   .   .   .   .   *** MOVE TURB KILLER TO TOP ***
 .   .   .   .   .   IF (KILLER.NE.0 .AND. I.NE.UPPER) THEN
 .   .   .   .   .   .   *** TURB KILLER TO TOP OF TURBS
 .   .   .   .   .   .   LOOP FOR (J=I+1,UPPER)
 .   .   .   .   .   .   .   IF (NODTYP(J).EQ.2) THEN
 .   .   .   .   .   .   .   .   CALL ROTATE(I,J,1)
 .   .   .   .   .   .   .   .   EXITLOOP (P)
 .   .   .   .   .   .   .   ENDIF
 .   .   .   .   .   .   ENDLOOP
 .   .   .   .   .   ENDIF
 .   .   .   .   .   *** PUT TURBS BELOW FIRST QUIET ***
 .   .   .   .   .   IF (I+1.LT.UPPER) THEN
 .   .   .   .   .   .   CALL ROTATE(NEWLOW+1,UPPER,UPPER-I+1)
 .   .   .   .   .   ENDIF
 .   .   .   .   .   EXITLOOP
 .   .   .   .   ENDIF
 .   .   .   ENDLOOP
 .   .   ENDIF
 .   .   IF (NEWLOW.NE.LOWER)CALL MERGE(LOWER,NEWLOW,UPPER)
 .   ELSE
 .   .   IF ((LEVEL.NE.1 .OR. .NOT.TSTING .OR. TSTRNK.EQ.0)
*.   .       .AND. (.NOT.PREORD .OR. LEVEL.NE.2)) THEN
 .   .   .   *** RETURN ONLY THE BEST  ***
 .   .   .   CALL ORDER2(LOWER,UPPER,1)
 .   .   ELSE
 .   .   .   *** FOR RANK TESTING RETURN THE ENTIRE LIST ***
 .   .   .   CALL ORDER2(LOWER,UPPER,UPPER-LOWER)
 .   .   .   IF (PREORD .AND. LEVEL.EQ.2) THEN
 .   .   .   .   LOOP FOR (I=LOWER,UPPER)
 .   .   .   .   .   IF (NODTYP(I).EQ.0) NODTYP(I) = 1
 .   .   .   .   ENDLOOP
 .   .   .   ENDIF
 .   .   ENDIF
 .   .   IF (NODTYP(LOWER).EQ.0) NODTYP(LOWER) = 1
 .   ENDIF
 ENDIF
```

October 1983

Target Module Generated by OVERDRIVE:

```
      IF (LEVEL.GE.VPLYMN) GO TO 91
        CALL ORDER2(NEWLOW,UPPER,UPPER-NEWLOW)
        IF (NEWLOW.NE.LOWER) CALL MERGE(LOWER,NEWLOW,UPPER)
      GO TO 95
91      IF (LEVEL.EQ.VPLYMX) GO TO 96
          CALL ORDER2(NEWLOW,UPPER,UPPER-NEWLOW)
          IF (VALUEF(NEWLOW).LT.0) GO TO 98
          IF (VALUEF(UPPER).GT.0) GO TO 98
            I99998=UPPER
            I = NEWLOW
            GO TO 912
910         I=I+(1)
912         IF (I.GT.I99998) GO TO 911
              IF (VALUEF(I).GT.0) GO TO 913
                IF (KILLER.EQ.0) GO TO 914
                IF (I.EQ.UPPER) GO TO 914
                  I99997=UPPER
                  J = I+1
                  GO TO 918
916               J=J+(1)
918               IF (J.GT.I99997) GO TO 917
                    IF (NODTYP(J).NE.2) GO TO 919
                      CALL ROTATE(I,J,1)
                      GO TO 917
919               GOTO 916
917             CONTINUE
914             IF (I+1.GE.UPPER) GO TO 924
                  CALL ROTATE(NEWLOW+1,UPPER,UPPER-I+1)
924             GO TO 911
913           GOTO 910
911         CONTINUE
98          IF (NEWLOW.NE.LOWER)CALL MERGE(LOWER,NEWLOW,UPPER)
          GO TO 932
96          IF (.NOT.((LEVEL.NE.1 .OR. .NOT.TSTING .OR. TSTRNK.EQ.0) THEN
     *       )) GO TO 933
            IF (.NOT.PREORD) GO TO 934
            IF (LEVEL.EQ.2) GO TO 933
934           CALL ORDER2(LOWER,UPPER,1)
            GO TO 937
933           CALL ORDER2(LOWER,UPPER,UPPER-LOWER)
              IF (.NOT.PREORD) GO TO 940
              IF (LEVEL.NE.2) GO TO 940
                I99996=UPPER
                I = LOWER
                GO TO 943
941             I=I+(1)
943             IF (I.GT.I99996) GO TO 942
                  IF (NODTYP(I).EQ.0) NODTYP(I) = 1
                GOTO 941
942           CONTINUE
940         CONTINUE
```

```
937         IF (NODTYP(LOWER).EQ.0) NODTYP(LOWER) = 1
932      CONTINUE
95       CONTINUE
```

October 1983

FORTRAN I/O LIBRARY

This section incorporates a summary of the methods of FORTRAN input/output under MTS, but does not attempt to provide a detailed description of the FORTRAN I/O statements. A more comprehensive description of the FORTRAN I/O statements is available in the IBM publications, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515, and VS FORTRAN Application Programming: Language Reference, form GC26-3986.

The FORTRAN I/O routines described in this section are automatically invoked when the FORTRAN program executes one of the I/O statements (READ, WRITE, PAUSE, ENDFILE, etc.) and they control the transfer of data between internal storage and I/O devices. In addition, the routines print error diagnostics and control error recovery when an I/O error occurs and when a program interrupt occurs.

The FORTRAN I/O routines are not the only way a FORTRAN program can perform an I/O operation. There are several subroutines available in MTS which will perform I/O and do not invoke the routines described here. This section describes only the FORTRAN I/O routines invoked by the FORTRAN I/O statements or the errors described above.

It is not advisable to combine the various methods of I/O in one program. If the FORTRAN I/O statements and control commands available through the FORTRAN I/O subroutine FTNCMD are used on a certain unit, then only the FORTRAN I/O facilities should be used on that unit. One should not mix the MTS I/O facilities with FORTRAN I/O facilities as unpredictable results may occur.

The FORTRAN I/O library has certain features which are not found in standard versions of FORTRAN IV. The library routines were written to reflect a terminal-oriented environment. Therefore, error messages are relatively self-explanatory and error recovery is possible. The "FATAL FORTRAN ERROR" that used to plague many users has been replaced with more precise diagnostics so that the possible cause of the error can be determined. In conversational mode, execution of the program may be resumed after an error.

A FORTRAN program must interface with the operating system for all communication. The operating system imposes certain restrictions on the FORTRAN program, thus making it sometimes difficult to transport a program from one installation to another. Programs written elsewhere may need some modification before they will execute properly in MTS (if the other installation also uses MTS as its operating system, the conversion effort should be minimal). Programs written to run under MTS, but intended for distribution to non-MTS installations, should be

designed so that the extended facilities provided by MTS are not required for proper execution.

Some of the features described are available at other installations. All of the facilities of the FORTRAN I/O library are available at other MTS installations, though possibly in a different form. However, many of these facilities are not available at non-MTS installations. In particular, the descriptions in this section of error handling, NAMELIST, and logical unit defaults are applicable only to the current implementation of the FORTRAN I/O library at the University of Michigan.

The information provided in this section does not apply to WATFIV (the Waterloo FORTRAN compiler). For information regarding input/output with WATFIV, see the section "WATFIV" in this volume.

Use of some of the direct-access features is restricted to programs running under MTS.

The FORTRAN input/output statements control, in a FORTRAN program, the transfer of data between internal storage and an input/output device, such as a reader, printer, magnetic tape unit, or disk storage.

LOGICAL UNIT ASSIGNMENTS

Specific file or device names are not usually referenced within programs. Instead, reference is made to logical I/O units, which are represented by numbers that denote a general input and/or output facility. This method of referencing was developed so that individual programs could be independent of specific files or devices.

FORTRAN recognizes units 0-99 as legal FORTRAN unit numbers (Data Set Reference Numbers). These units are normally mapped into MTS logical I/O units 0-99. However, it is important to note that FORTRAN I/O units and MTS logical I/O units are not necessarily equivalent; the relationship may be altered by the programmer (see below).

MTS Unit Assignments

The relationship between an MTS logical I/O unit and a file or device can be established in several ways.

For example, at the MTS command level,

```
$RUN -LOAD 1=DATA 18=*SINK*
```

establishes the equivalence of the MTS logical I/O units 1 and 18 to the file DATA and to the current output device, respectively.

In general, MTS logical I/O unit assignments are specified on the $RUN command as follows:

```
$RUN object munit=FDname
```

where

munit is the MTS logical I/O unit (0-99, SCARDS, SPRINT, SPUNCH, SERCOM, or GUSER).
FDname is the name of an MTS file or device.

Logical I/O unit assignments can also be made during execution of the FORTRAN program by a subroutine call to FTNCMD.

```
CALL FTNCMD('ASSIGN 1=DATA;')
```

will assign or reassign MTS unit 1 to the file DATA. See the subsection "FORTRAN Command Language Monitor" for a complete description of the FTNCMD calling sequence.

If logical I/O units are not explicitly assigned on the $RUN command, MTS will default the following MTS logical I/O units.

```
SCARDS to *SOURCE*
SPRINT to *SINK*
GUSER  to *MSOURCE*
SERCOM to *MSINK*
SPUNCH to *PUNCH* (if in batch mode and if the CARDS global
                   parameter has been specified on the $SIGNON
                   command).
```

If not explicitly assigned by one of the above methods, the FORTRAN I/O library will default the following MTS logical I/O units:

```
Unit 5 to *SOURCE*
Unit 6 to *SINK*.
```

The other units are not assigned by default; thus an error condition will be produced if they are referenced but not explicitly assigned.

The FORTRAN I/O Command Language Monitor uses SERCOM for error-message printing and GUSER for user-prompting.

FORTRAN Unit Assignments

In order to perform any I/O operations in MTS, the FORTRAN units must be associated with MTS logical I/O units, namely, SCARDS, SPRINT, SPUNCH, GUSER, SERCOM, and 0-99, or to an MTS file/device. Normally, the FORTRAN units 0-99 are interfaced to the MTS units 0-99, 5 to SCARDS if the MTS unit 5 is not assigned on the $RUN command, 6 to SPRINT if not assigned.

The user may alter the relationship between FORTRAN unit numbers and MTS logical I/O unit numbers. For example, the user can do this by executing in the program, prior to using the logical I/O unit, the following:

```
CALL FTNCMD('EQUATE funit=munit;')
```

where

funit is a FORTRAN logical I/O unit number from 0-99.
munit is the MTS logical I/O unit on which the I/O operation is to occur.

For example, assume that a FORTRAN program used FORTRAN unit 99 for error messages and FORTRAN unit 56 for input. The following two statements are inserted in the program to map FORTRAN units 99 and 56 to SERCOM and MTS logical I/O unit 3, respectively.

```
CALL FTNCMD('EQUATE 99=SERCOM;')
CALL FTNCMD('EQUATE 56=3;')
```

A user can associate the FORTRAN I/O units 0-99 to an MTS file or device directly. In this case, I/O is performed to the file or device without using an MTS unit.

```
CALL FTNCMD('ASSIGN 20=FYLE;')
```

FORTRAN I/O ACCESS

The READ and WRITE statements are the two basic I/O statements in a FORTRAN program. The READ statement is used to transfer data from an external device such as a card reader, disk drive, or tape drive to internal storage. The WRITE statement is used to transfer information from internal storage to an external device such as a printer, disk drive, or tape drive.

The READ and WRITE statements will specify either sequential or direct access I/O. Sequential I/O implies that the next record in sequence is accessed. Direct access or indexed I/O means that the location of the next line to be read or written is specified in the I/O statement. With indexed I/O, the next record accessed is not necessarily the next record in sequence. With VS FORTRAN, direct-access I/O is specified in I/O statements by the REC specifier.

The two types of I/O access, sequential and direct, determine which record is read and where the results are written. The format of an I/O statement determines how the record is read or written. For example, the format on a formatted READ statement defines how many records are to be read, the part of the record that is to be ignored and specifies how the data are to be converted before being placed in internal storage.

If no format is specified, there is a one-to-one relationship between the values in the record and internal storage. No conversion is performed. The various forms of FORTRAN I/O conversions are described in the following section.

And now let us define the sequential and direct access methods.

Sequential I/O

Sequential I/O is the accessing of records in a series. For sequential files, tapes, and unit record equipment, records are accessed as they appear in sequence.

The full form of the READ/WRITE operation for sequential I/O is:

FORTRAN G and H:

```
READ (unit,format,END=m,ERR=n) list
WRITE (unit,format,ERR=n) list
```

VS FORTRAN:

```
READ (UNIT=unit,FMT=format,END=m,ERR=n,IOSTAT=status) list
WRITE (UNIT=unit,FMT=format,ERR=n,IOSTAT=status) list
```

where

unit	is the FORTRAN I/O unit number.
format	is the format.
m	is the statement label to which the program transfers if an end-of-file condition occurs.
n	is the statement label to which the program transfers in event of an I/O error.
status	is the I/O status, or error or end-of-file condition.
list	is a list of FORTRAN variables (possibly null) to which values are assigned.

For line files, the sequence of records accessed is determined by the range of the beginning and ending line numbers, and the increment, specified in the FDname on the $RUN command. If the line file is not rewound or positioned by any of the direct access methods (see the subsection "Direct Access I/O"), the first record accessed will correspond to the initial line number in the line number range specified in the FDname. If the initial line number was not specified, the default line number 1 is used. When the line number increment is omitted from the $RUN command FDname, a sequential READ will always access the next record in the file, regardless of its line number. However, if a line number increment is specified, a sequential READ can access only those records that have line numbers that are offset from the starting line number by multiples of the increment.

For example, suppose the file DATA has lines numbered 1, 1.5, 2, 2.1, 4, 5, 6, and 7. The following table indicates the lines that would be accessed by sequential READ statements with various $RUN command FDname assignments.

$RUN FDname	Lines Accessed
DATA	All lines
DATA(1,,1)	1 2 4 5 6 7
DATA(1,,2)	1 5 7
DATA(2,,2)	2 4 6
DATA(1,5,.5)	1 1.5 2 4 5

A sequential WRITE to an MTS line file always uses the line number increment, or the default of 1 if the increment was not specified, when writing the next record to the file. Using the same example, if we write sequentially starting at line number 1, new lines 1, 2, 3, 4, 5, 6, and 7 would be generated. This would leave old lines 1.5 and 2.1 intact, and would add line 3 to the file.

End-of-File Exit

Using a sequential READ statement allows a user to intercept the end of data return. This option is not available with direct-access READ.

Reading an end-of-file with no END= exit provided on the READ statement causes a return to MTS. The program may be restarted with a $RESTART command. Execution resumes in the FORTRAN Monitor command mode. The user who decides to continue execution must either enter "RETURN" or "RETURN SKIPIO".

When an end-of-file condition is encountered, control is transferred to the statement specified on the END= exit, provided there was an exit specified. This allows the user to continue execution of the program even after the end of data has been detected.

If the user issued the MTS command

```
$SET ENDFILE=ON
```

an $ENDFILE line detected in the data stream is treated as an end-of-file condition as described above.

The Error Exit

The ERR= exit is available for all error conditions, including conversion errors. When an error is detected during the READ or WRITE operation, control is transferred to the statement specified in the ERR= exit.

If the ERR= exit is not specified, a program is usually terminated unless the SIOERR exit routine is active. The SIOERR exit routine should not, however, issue any FORTRAN I/O statement before returning.

The following statements demonstrate how the ERR= exit can be used in a FORTRAN program.

```
C
C     Read in the data cards.
C     Ignore those that have invalid data on them.
C
      K=0
1     K=K+1
      READ (5, 900, END=200, ERR=300) N, I, J
       .
       .
       .
      GO TO 1
200   WRITE (6,905)
      STOP
300   WRITE (6,910) K
      GO TO 1
900   FORMAT (3I10)
905   FORMAT (' END OF DATA ENCOUNTERED.')
910   FORMAT (' RECORD', I5, ' SKIPPED.')
      END
```

In the above example, the FORTRAN I/O routines print the error message if ERRMSG is ON before transferring control to the statement 300. Since the default of ERRMSG is OFF, the user may enable the printing of error messages by inserting the following statement:

```
CALL FTNCMD('SET ERRMSG=ON;')
```

To disable all the ERR= exits, a user may include the following call in a FORTRAN program:

```
CALL FTNCMD('SET ERR=OFF;')
```

When error exits are disabled by the FTNCMD call as above or if error exits are not specified, normal error processing is invoked when an error occurs. In conversational mode, the error message is printed and, if possible, the user is prompted to answer. In batch mode, the error message and diagnostics are printed and the execution is terminated.

Direct Access I/O

In direct access I/O or indexed I/O, the location, or index, of the next line to be read or written can be specified in the I/O statement. Direct access I/O can be performed only on line or sequential files.

Direct Access to Line Files:

There are three indexed input/output statements: READ, WRITE, and FIND. The READ and WRITE statements transfer data into and out of main storage. A FIND statement does not transfer data, but causes indexing of the next READ/WRITE statement. The general forms of these statements are:

```
READ (unit'index,format) list                (FORTRAN G and H)
WRITE (unit'index,format) list               (FORTRAN G and H)
FIND (unit'index)                            (FORTRAN G and H)

READ (UNIT=unit,REC=index,FMT=format) list   (VS FORTRAN)
WRITE (UNIT=unit,REC=index,FMT=format) list  (VS FORTRAN)
```

The following points should be noted for indexed I/O operations:

(1) The index specified is the MTS line number times 1000. For example, to reference line 4.7 in a file, the index is 4700.

(2) On any indexed I/O operation, the index must be in the range -2,147,483,648 to 2,147,483,647. An index specified explicitly as a constant must be in the range 0 to 16,777,215 (i.e., 16^6-1). Thus,

```
FIND(4'16800000)
```

will not work, whereas

```
NREC=16800000
FIND(4'NREC)
```

will work.

(3) If an index given in a READ statement represents a nonexistent line, an end-of-file condition is generated, even if the line was not the last in the file. Note that the END=n exit cannot be specified in an indexed READ, thus it is recommended that a FIND followed by a sequential READ be used if it is possible that the line specified by the index does not exist.

(4) If an indexed READ or WRITE statement requires the reading or writing of more than one record, only the first record will be read or written indexed; any subsequent records will be obtained sequentially. For example, the following statements would read line number 10 and the next sequential line in the file that was attached to unit 5.

```
      READ(5'10000,56)A,I
56    FORMAT(G15.8/I10)
```

Note: The "next" line in the sequential sense will be governed by the increment specified in the $RUN command FDname.

(5) If the next I/O statement after a FIND is an indexed READ or WRITE rather than a sequential READ or WRITE, the index specified in the READ/WRITE operation will override that specified in the FIND.

(6) The DEFINE FILE statement may be used to define a particular I/O unit attached to an MTS line file. If this is used, all of the direct access I/O statements will behave as defined in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

Direct Access to Sequential Files:

Direct access to sequential files has been implemented in two ways:

(1) Using the IBM DEFINE FILE statement:

If the DEFINE FILE statement has been used to define a particular I/O unit, any of the direct access I/O statements READ, WRITE, and FIND will behave as defined in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

(2) Using MTS NOTE-pointers:

Pseudorandom access can be performed on sequential files from a FORTRAN program via subroutine calls to NOTE and POINT. A description of these routines is available in MTS Volume 3.

If the program is to be run exclusively at an MTS installation, it is recommended that direct access to line files be used. If the program is to be transported to non-MTS installations, it is recommended that the DEFINE FILE statement (for FORTRAN G and H) or the OPEN statement (for VS FORTRAN) for direct-access sequential files be used.

FORTRAN I/O CONVERSIONS

The FORTRAN READ and WRITE statements invoke one of four basic forms of I/O conversion. The forms are given below:

(1) Formatted READ/WRITE

```
READ (unit,fmt) list
WRITE (unit,fmt) list
```

(2) Unformatted READ/WRITE

```
READ (unit) list
WRITE (unit) list
```

(3) NAMELIST READ/WRITE

```
READ (unit,naml)
WRITE (unit,naml)
```

(4) List-Directed READ/WRITE

```
LOGICAL*1 fmt(1)/'*'/   (FORTRAN G and H)
READ (unit,fmt) list
WRITE (unit,fmt) list

READ (unit,*) list      (VS FORTRAN)
WRITE (unit,*) list
```

Formatted I/O means that the programmer has supplied, along with the I/O list of what to read or write, a format describing how the data are to be read or written. Using a formatted READ generally means that the data are converted according to the corresponding format specifications before being placed in internal storage. It is possible to use a formatted READ statement and essentially use a free-format method of input. The pseudofree-format input method and the list-directed I/O method are described in this section.

With unformatted I/O, the data are transmitted without control of a format. There is no conversion from internal representation to external representation and vice versa. An unformatted WRITE operation will take the internal representation of a number and write that on the external device. There is no conversion to character form, so the value printed is unreadable. Because there is no conversion, unformatted I/O is faster than formatted I/O. An unformatted WRITE operation is typically used in an application where the output is to be used as input to another program without being examined by a programmer.

Instead of an I/O list, NAMELIST I/O specifies a NAMELIST dictionary to define the variables which may be read by a NAMELIST READ statement and are written by a NAMELIST WRITE statement. NAMELIST I/O consists of keyword-type entry and output, where an "=" is used to separate the variable names from their values. Conversion is performed according to the type of the variable. For example, if the variable is declared to be INTEGER, then an integer value is expected on input and written on output.

With list-direct I/O, the data values for both input and output are separated by a comma, one or more blanks, or the end of the record. This is a free-format style with no constraints to specific columns. The conversion is performed according to the type of the variable being read or written. For example, if the variable in the list is declared as REAL, then a real value is read or written. See the section "List-Directed I/O" for further details.

The method used depends entirely on the effect desired. For input, one of the free-format methods described under formatted I/O is probably easiest, since it frees the user from column restrictions. For output,

if output in tabular form is desired, a formatted WRITE statement must be used. If the output does not have to look pretty, perhaps list-directed output (described under the subsection "Formatted I/O") or NAMELIST output will suffice.

Warning: The FORTRAN I/O Library, the FORTRAN G DATA statement, and the FORTRAN H DATA statement do not provide consistent conversions from the external decimal representation to the internal hexadecimal representation for floating-point numbers. The FORTRAN G compiler generates internal numbers which may be in error by several units in the last place when compared to results generated by the FORTRAN I/O Library. The FORTRAN H compiler provides a more accurate conversion, but the results are not guaranteed to be identical with those produced by the FORTRAN I/O Library. Therefore, in some types of computations, these discrepancies may have a significant impact. In general, it is not a good programming practice to make exact comparisons of floating-point numbers. Rather, the user should check for a value falling within a narrow range; e.g., IF(ABS(A-2.0).LT.EPSILN) where the value of EPSILN is chosen to be an acceptable tolerance.

Formatted Conversion

When data are transmitted under control of a format statement, a conversion is made so that the input is in a form which satisfies the needs of machine representation, or so that the output is more readable to the programmer. For input operations, the character representation of the data is converted to the appropriate internal machine representation. For output, the machine representation is converted to character notation.

Codes specified in the format determine how the data are to be converted on an I/O operation. For example, I or G format is used for integer conversions and F, E, D, or G formats are used for floating-point conversions. Q format is not supported.

The length of the record and the form of the data fields within the record are specified by the I/O list and the format statements. A format may be either a compiled format or an object-time format. Formats that correspond to the FORTRAN FORMAT statement are called compiled formats. Their structure is unalterable during execution. Alternatively, the user may choose to read a format into an array as data during program execution, or in some other way dynamically construct a format in an array. Such formats are called object-time formats or variable formats because their structure is defined at object or execution and may be altered. For example,

```
      DIMENSION A(100)
      LOGICAL*1 FMT(80)
C READ IN THE NUMBER OF VARIABLES TO BE PROCESSED.
      READ(5,100)INUM
```

```
  100 FORMAT(I2)
C READ IN THE FORMAT FOR THE DATA.
      READ(5,200)FMT
  200 FORMAT(80A1)
C READ IN THE DATA USING THE ABOVE FORMAT.
      READ(5,FMT)(A(I),I=1,INUM)
```

Free-format I/O means that the READ and WRITE operations are freed from format restrictions. Instead of being entered within rigid columns boundaries, the data are entered such that adjacent data fields are separated by a delimiter. There is still a conversion from external form to internal form and vice versa.

Two forms of free-format I/O are described in this section. One is called semifree-format input because although a format is supplied with the READ statement, the column restrictions need not be adhered to. The second form is called list-directed I/O and applies to both input and output. Also available, although not part of FORTRAN I/O routines, is the FORTRAN-callable free-format input routine FREAD. See the section "FREAD: A Free-Format Input Subroutine" in this volume for details.

Semifree Format Input:

Free-format input simplifies data entry in that the user is freed from format-statement restrictions. Rather than having to enter data within rigid column boundaries, the user simply needs to make sure that adjacent data fields are separated by a delimiter. For information regarding this facility, the user is referred to the section "FREAD: A Free-Format Input Subroutine" in this volume.

FORTRAN I/O does, however, have a semifree-format facility. On input, any field to be converted according to a format code (other than A format) is scanned for one of four special delimiters: comma, slash, quote, or semicolon. If none of these separators are found, the conversion is performed according to the format specifications. If a separator is encountered in the field and the format specifies that there are to be "d" significant digits after the decimal point and the decimal point is not explicitly specified in the input field, the decimal point is assumed to be to the right of the last digit in the field.

For example, if the format code being used is F10.3,

```
456.3,     sets the variable to 456.300
4563,      sets the variable to 4563.000
3,         sets the variable to 3.000
```

If a comma is found, the field width is changed to terminate with the last character before the comma. A slash or a quote functions in the same manner as a comma. If a semicolon is found, processing proceeds as if a comma had been found. However, after the current

field has been processed, the read operation is terminated. Accordingly, any remaining list items will remain unchanged.

If the resulting field width is zero (which would be the case if there were two adjacent commas), the list item is left unchanged.

This method of semifree-format input will work only if the delimiter occurs within the field width. Therefore, the column width must be large enough to encompass the delimiter.

Example:

```
C PROGRAM TO ILLUSTRATE THE USE OF
C SEMI-FREE FORMATTED INPUT
 10   READ(5,100,END=300)A,N,M
 100  FORMAT(F20.8,I5,I3)
      WRITE(6,200)A,N,M
 200  FORMAT(F10.5,2I6)
      GOTO 10
 300  STOP
      END
```

The results of this sample program follow. The odd-numbered lines represent input, and the even-numbered lines represent output.

```
(1)   1.,5,6;
(2)     1.00000     5     6
(3)   1.5;
(4)     1.50000     5     6
(5)   ,111113,
(6)     1.50000 11111     3
(7)   ,,2
(8)     1.50000 11111   200
```

The first line of input, along with the resulting output, should be self-explanatory. The second set of input values illustrates premature termination of I/O processing, accomplished here by using a semicolon. In the third set of input values, the first operand is omitted and remains unchanged. The second operand fills the entire field and consequently, the field is not terminated by the final comma. Note, that if the line had been given as

```
,11111,3
```

the third list item (variable M) would not have been changed, since the additional comma would not have been found until the I3 field was scanned. The final set of input values illustrates the usefulness of a final comma or semicolon. If either of these had been used, it would have been clear that the intended value of M was 2 rather than 200.

FORTRAN G and H users can avoid the problem posed by the fourth set of values in this example by using the subroutine FCVTHB. The calling sequence is:

```
CALL FCVTHB(arg)
```

where "arg" is the location of the fullword argument. When arg=0, all blanks on numeric input are assumed to be zeros. This is the normal FORTRAN mode. When arg does not equal zero, all blanks are usually ignored. However, a field consisting entirely of blanks has a value of zero. Calling FCVTHB at the beginning of the program with a nonzero argument will eliminate the problem illustrated by the fourth set of values. The default processing is equivalent to a call to FCVTHB with arg=0. The processing of blanks during program execution can be changed at any time by calling FCVTHB. Alternatively, VS FORTRAN users should use the BZ and BN edit specifiers in the FORMAT statement. They also can use the BLANK specifier (BLANK=ZERO or BLANK=NULL) in the OPEN statement.

List-Directed I/O

If the first character of a variable format is an asterisk (*), list-directed I/O will be used. The following program segments invoke list-directed I/O:

FORTRAN G and H:

```
LOGICAL*1 FMT(1)/'*'/
READ (5,FMT) A,B,I,K
WRITE (6,FMT) A,B,I,K
```

VS FORTRAN:

```
READ (5,*) A,B,I,K
WRITE (6,*) A,B,I,K
```

The features of list-directed I/O are described in detail in the IBM publications, IBM System 360 and System 370 FORTRAN IV Language, form GC28-6515, and VS FORTRAN Application Programming: Language Reference, form GC26-3986. Note that this feature is not available using an '*' in place of the format specification in the READ and WRITE statement as described in the IBM publication (e.g., READ(5,*) is illegal except for VS FORTRAN). A brief description of the list-directed I/O features follows for FORTRAN G and H:

(1) List-directed I/O is sometimes referred to as stream input or output. As many records as are required to satisfy the I/O list will be read or written. The data values are separated by a comma, one or more blanks or the end of record. The obvious advantages are that the data values need not be punched in specific columns and each data

record could be punched differently than its predecessor. The list-directed WRITE statement is very useful if the user is debugging his program and would like to print the variables at various stages of the calculations. It provides a quick method of inserting WRITE statements, without having to provide a FORMAT for each one.

(2) Conversion is performed on input and output according to the type of the variable. If the variable is declared to be INTEGER, then only integer values will be accepted on input.

(3) Consecutive commas on input cause the associated list variable to remain unchanged. For example, if the following line is read by the above program segment:

```
123.,,456 78
```

the variable A is set to 123., B retains the value it had before the READ statement, and I and K are set to 456 and 78 respectively.

(4) A slash (/) may be used to terminate the input. Items in the I/O list which are not satisfied remain unchanged. Thus, to change only the value of A in the above example,

```
34.5/
```

would set A to 34.5 and leave the remaining variables unchanged.

(5) Replication counts are allowed on input. The line

```
4*0
```

would zero the four variables in the I/O list.

(6) Literal constants may be read in but cannot be written using list-directed I/O. The characters are enclosed in primes (') and must not be longer than the element size, or the string will be truncated. For example, for an INTEGER variable, four characters should be enclosed in primes, e.g., 'ABCD'.

Options Available with Formatted I/O

A formatted READ or WRITE statement that specifies conversion according to a format code is controlled by the setting of the following options. Note that the only option which will affect list-direct I/O is the suppression of undefined variable checking.

(1) WRAPAROUND

If a formatted write produces a record that is longer than the device length, the IBM FORTRAN manual states that the record length will be truncated. Here, an alternative method to truncate has been developed and is called "wraparound." If a formatted write produces a record that is longer than the device length, the record is broken up and written on several lines. Wraparound is the default action. If truncation is preferred, wraparound can be disabled with the following call to FTNCMD:

```
CALL FTNCMD('SET WRAPAROUND=OFF;')
```

Wraparound acts as if a "/,1X," were inserted in the format immediately preceding the format specification to be used on the variable field that would have been truncated.

For example, given the following statements in a FORTRAN program:

```
      INTEGER I(35)
      WRITE (7,1) NUM, I
    1 FORMAT (1X, I2, 35A4)
```

If the unit 7 is attached to a line file, one line will be written because the device length of the file is 32767 characters. If the unit 7 is attached to a printer, wraparound will take effect because the device length for a printer is 133 (actually 132 print positions plus one carriage-control character), and the output length for the above WRITE statement is 143 characters. The above WRITE statement will write two lines of information using the following format:

```
(1X, I2, 32A4, /, 1X, 3A4)
```

If, on the other hand, the unit 7 was attached to *PUNCH*, a device with an 80-character length, the above WRITE statement would punch two cards with the following format:

```
(1X, I2, 19A4, /, 1X, 16A4)
```

Note that when wraparound is used, and if the records produced are to be read, the modified format shown above must be used for the READ.

A word of warning. Often strange results are obtained when T-formats are used on lines that are wrapped. If the user wants to use T-formats, he should make sure the length of the line being written is less than the device length. If not, the wraparound feature should be turned off.

(2) Undefined Variable Checking

If the value to be written is identically equal to a string of X'81', an error condition is generated and the value will be represented in the output buffer with a string of "U"'s. If this does occur, it means the variable to be written was either undefined or assigned value -2122219135 (INTEGER) or approximately -0.7E-76 (REAL). The undefined variable checking can be disabled with the following call to FTNCMD:

```
CALL FTNCMD('SET UVCHECK=OFF;')
```

(3) Mode Checking

The type of a variable should agree with the type of the format code used to read or write the variable. For example, an INTEGER variable should have an I or G format specified, and a REAL variable should have one of F, G, E, or D format codes specified.

If the types do not agree, a diagnostic is generated and the G-format is used by default. The type checking can be disabled with the following call to FTNCMD:

```
CALL FTNCMD('SET MODECHECK=OFF;')
```

The G format code cannot be used by default for character variables.

(4) Zero-Length Records

Zero-length records, caused by two or more consecutive slashes in the format, are changed to write one blank. This preserves printer-spacing if the output is written on a file. The following call to FTNCMD will disable this feature and write a true zero-length line.

```
CALL FTNCMD('SET NULLBLANK=OFF;')
```

Note that an indexed write operation of a zero-length line onto a file deletes the line from the file.

```
      WRITE (4'NREC, 43)
   43 FORMAT (I5)
```

The above WRITE statement will write a blank line at line NREC if NULLBLANK is set ON or delete the line number NREC from the file if NULLBLANK is set OFF.

(5) Suppressing Zeros on Output

The following call to FTNCMD will suppress printing of true zeros.

```
CALL FTNCMD('SET ZEROSUPPRESS=ON;')
```

The above causes zeros to be printed as blanks.

(6) Blank Input Detection

The following call will return any blank field encountered by a formatted READ statement as a -0.0.

```
CALL FTNCMD('SET MINUSZERO=ON;')
```

This feature works for REAL variables read with D, E, F, or G format. This feature is not available for INTEGER variables.

Unformatted Conversion

When data are transmitted without format control, there is a one-to-one correspondence between internal storage locations (bytes) and external record positions. One logical record is defined as the data corresponding to one I/O list in a FORTRAN READ/WRITE statement. Each logical record may consist of one or more physical records (or lines), depending on the length of the list, the number of bytes taken up by each list item, and the maximum size record allowed for the device.

Each physical record (or line) has two control words (8 bytes) which describe the physical record. Thus, if data in a line file (maximum record length = 32767 characters) are transmitted with no format control, a maximum of 32767 bytes is available for data bytes. Individual data elements are never split over two lines. For example, each line in a line file can hold up to 8189 fullwords (32756 bytes) of INTEGER*4 or REAL*4 data, 16379 halfwords (32758 bytes) of INTEGER*2 data, or 4094 doublewords (32752 bytes) of REAL*8 data.

Thus, an unformatted READ or WRITE can read or write several file lines in one operation. A BACKSPACE operation will always have the effect of positioning the pointer to the beginning of the logical record. For example, if a logical record is written with three physical records, a BACKSPACE will cause positioning to the beginning of the first of the three physical records.

The two control words that make up each physical record are automatically inserted by the FORTRAN I/O routines on a WRITE statement and are deleted on a READ statement. When performing an unformatted WRITE onto a magnetic tape, VS format (variable length, spanned) should be specified if the tape is labeled. U format (undefined length) should be specified if the user does not wish to disable tape blocking or if there is a mixture of formatted and unformatted records in a single file. Unformatted I/O cannot be used with ASCII-labeled tapes. When sending a tape to an OS/VS installation, each file may be either formatted or unformatted, but no file may contain both formatted and

unformatted records. Since the actual records conform to VS format, which is unblocked, tapes containing VBS-formatted files (especially from OS/VS installations) must first be deblocked into VS-format files so that they can be read by the FORTRAN I/O routines. For further information on the use of magnetic tapes, see MTS Volume 19, Tapes and Floppy Disks.

Advantages:

It is recommended that unformatted I/O be used when transferring data between programs, as it is faster than using format control. Unformatted I/O is about 6 times faster than I/O using a typical F-type numeric formatted conversion and is about 1.5 times faster than I/O using A format conversion.

Unformatted I/O is best utilized where intermediate results, which are to be read in again for further processing, are produced. The printed results obtained by $LISTing records from a file or tape that contains unformatted data are usually unintelligible since such records contain the internal machine representation of the data. Unformatted files can be handled only by some other program (in FORTRAN or another programming language). If the unformatted data are to be read by a FORTRAN program, or a program in another language, the corresponding I/O list must be compatible with the I/O list in the write operation. If a logical record is generated from a list of n words during output to a file, then a subsequent input statement attempting to read this record must have a list of not more than "n" words.

Unformatted Records:

An unformatted WRITE will write one logical record which is made up of one or more physical records (or lines).

Each physical record has 2 control words, each occupying 4 bytes. The control words are called the BCW (Block Control Word) and the SCW (Segment Control Word).

BCW	2 byte length	2 bytes unused	
SCW	2 byte length	1 byte flag	1 byte unused

where:

BCW length specifies the number of bytes in the physical record, including the BCW and SCW.

SCW length specifies the number of bytes in the physical record, excluding the BCW, i.e., SCW length = BCW length - 4 bytes.

SCW flag is used to describe the order of physical records within a logical record. This byte will be set to:

x'00' if the logical record is made up of only one physical record.
x'01' if this is the first physical record of many.
x'02' if this is the last physical record.
x'03' if this is an intermediate physical record, that is, it is neither the first nor the last.

These two control words are used to allow the FORTRAN programmer to output data without knowing the maximum physical record length of the device. It can even exceed 32767.

Direct Access I/O and Unformatted Records:

Caution must be exercised when using direct access I/O on defined files and unformatted reading or writing on the same file. Each unformatted WRITE may produce more than one line. Unless the number of lines written for each WRITE statement is less than the line number increment used by the program in the output of successive logical records, there will be overwriting of lines already written and the file may become unreadable.

NAMELIST Conversion

NAMELIST provides a means for reading and writing data without including the I/O list and format specifications. The NAMELIST specification defines a name to refer to a particular list of variables. The conversion performed on output, and usually on input, is entirely dependent upon the type of the variable, e.g., integer conversion is performed for INTEGER variables and floating-point conversion is performed for REAL and COMPLEX variables.

NAMELIST input has been implemented as defined in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515. An alternative method of NAMELIST input "free-format" was added and defined below.

NAMELIST output was implemented as described in the above IBM manual. The routine will write as many records as are needed to satisfy the NAMELIST declaration statement.

NAMELIST Input:

There are two forms of NAMELIST input, IBM format and free-format. The user may require NAMELIST input to be in IBM format by an appropriate call to FTNCMD:

```
CALL FTNCMD('SET NAMEFMT=IBM;')
```

Alternatively, if the FREE format is used, the user may include the call:

```
CALL FTNCMD('SET NAMEFMT=FREE;')
```

If neither is specified, the format of input will be determined as follows:

(1) IBM Format

If the second character in the first record is an ampersand, (&) immediately followed by the NAMELIST name, IBM format is used. The procedures for using IBM format are defined in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

(2) Free-Format

This format is used if an ampersand (&) is not present in column 2 of the first record. This form of input is much easier to use both in batch and conversational mode since the rules are not as rigid as the IBM specifications. The following points describe free-format for NAMELIST.

(a) The ampersand (&), followed by the NAMELIST name, must not appear on the input record. The first item on the record should be a variable, as defined in the NAMELIST declaration statement, equated to some value.
(b) The input stream generally consists of one record unless an MTS continuation character is present in the last column of the record. If a continuation character is present in the record, another record will be read. The MTS continuation character is a hyphen (-) unless otherwise specified. The continuation character can be changed at the MTS level with the command $SET CONTCHAR.
(c) Both blanks and commas are recognized as delimiters.
(d) Blanks preceding a delimiter are ignored. This is a major difference between IBM format and free-format. The statement

```
I=12 ,
```

denotes 120 in IBM format, while in free-format it denotes 12.
(e) The data stream is ended by &END or by the physical end of the record.
(f) Null (zero-length) lines are ignored.

NAMELIST Output:

NAMELIST output is defined in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

Options Available with NAMELIST:

(1) Undefined Variable Checking

Undefined variables will be set to "U"'s in the output buffer. This may be disabled with the following call to FTNCMD:

```
CALL FTNCMD('SET UVCHECK=OFF;')
```

(2) Suppression Of NAMELIST Name

The NAMELIST name (the name that occurs in the NAMELIST declaration statement) is written as the first line and &END is written as the last line of NAMELIST output. This may be suppressed with the following:

```
CALL FTNCMD('SET NAMEOUT=OFF;')
```

NAMELIST Notes and Restrictions:

(1) Hexadecimal Input

The character Z is used to transmit hexadecimal data on input. If the first character after the equal sign is a Z, hexadecimal conversion is performed regardless of the type of variable.

```
A=Z6B
```

(2) Repetition Counts

Repetition counts are valid on input.

```
A=3*'*'
```

sets A=***, where A is INTEGER*4

```
ARRAY(3)=4*0.0
```

sets elements 3 through 6 of ARRAY to zero.

(3) Literal Input

Literal information and Hollerith information are accepted on input. The remainder of the word and/or array is padded with blanks.

```
ARRAY=5HHELLO or ARRAY='HELLO'
```

set the first 5 characters of ARRAY to HELLO; the remaining characters in the array are set to blanks.

(4) Null Fields

A null field (two commas with no intervening characters) is considered an error.

```
A=, and A=1,,
```

are both errors, whereas

```
A=1 ,
```

is not an error.

(5) Complex Input

Complex variables may have values in parentheses.

```
COMPLEX A(2)
A=(1.0,2.0),(1.0,2.0) or
A=2*(1.0,2.0) or
A=1.0,2.0,1.0,2.0
```

A repetition character within parentheses is considered an error; i.e.,

```
A=(2*1.0)
```

is an error. This should be written as:

```
A=2*1.0
```

(6) Logical Input

When assigning a value to a logical variable, the field must consist of optional blanks followed by a "T" or "F" followed by optional characters. ".TRUE." and ".FALSE." are also accepted.

(7) Logical Output

On logical output the field will contain the logical operator T if the variable was nonzero, and the logical operator F if it is zero.

OTHER FORTRAN I/O STATEMENTS

The REWIND Statement

A REWIND statement can be used at any time to reposition a tape or to reset a file so that the next READ or WRITE statement acts at the beginning of the file or device. In a line file the beginning is the starting line number as defined when the unit was assigned, or the line number 1 by default. On a tape, the beginning is defined at the start of the first file of the tape. It should be noted that a REWIND statement causes rewinding to the beginning of the currently active element of a concatenation of FDnames. Only if the first element is being processed will REWIND cause a complete rewind.

The BACKSPACE Statement

The FORTRAN statement BACKSPACE n works on any input device (n is a FORTRAN unit). The effect of executing one BACKSPACE statement, followed by a sequential READ, is to reread the current logical record. In general, for files and tapes, the effect of executing n+1 BACKSPACE statements, followed by a sequential READ, is to read the current minus nth logical record.

(1) Executing a BACKSPACE statement on a file is the same as performing a sequentially backward READ with no data transmitted. For example, assume a file contains records at the following line numbers:

```
1  1.5  2  2.5  3  3.5  4  4.5
```

If line number 4.5 has just been read (the 8th record in the file) and the following statements are executed,

```
      DO 67 I=1,3
67    BACKSPACE UNIT
      READ(UNIT,FMT)LIST
```

the sequential READ will read line number 3.5 (the 6th record in the file).

Note: BACKSPACE is not predictable if the file has been assigned with a negative line increment or with the @BKWD modifier.

(2) When a BACKSPACE statement is issued for a tape, the CONTROL subroutine is used to issue a BSR command to skip backward past one logical record. For more information on backspacing tapes, see MTS Volume 19, Tapes and Floppy Disks.

(3) When a BACKSPACE statement is issued on any other device (e.g., a card reader), the last record in the buffer is reused. However, if the last record was unformatted, an error condition is generated. When a device such as a card reader is backspaced, the last record is reread, thus providing a method for rereading records using different formats.

Note: This applies only to the last record read, but BACKSPACE cannot be used to reread any previous records.

The BACKSPACE statement should not be used for list-directed records.

The ENDFILE Statement

When an ENDFILE statement is executed on a magnetic tape, the CONTROL subroutine is used to issue a WTM command on the tape. For more information on tapes, tape marks, and the CONTROL subroutine, see MTS Volume 19, Tapes and Floppy Disks.

An ENDFILE statement applied to any other type of device is ignored.

The PAUSE Statement

The forms of the PAUSE statement are

```
PAUSE
PAUSE n
PAUSE 'message'
```

where "n" is an integer from 0 to 99999. The PAUSE number or message is printed on *MSINK*, i.e., the terminal for conversational mode and the printer for batch mode. In batch mode the program resumes execution after the PAUSE statement. In conversational mode, control is returned to MTS. The program can then be restarted with a $RESTART command.

The STOP Statement

The forms of the STOP statement are

```
STOP
STOP n
STOP 'message'   (VS FORTRAN only)
```

where "n" is an integer from 0 to 99999. The STOP number or message is printed on *MSINK* and execution of the program is terminated. If "n" is zero or omitted, no message is printed.

The OPEN Statement

An OPEN statement may be used in VS FORTRAN to connect a file to a FORTRAN I/O unit. See the IBM publication, VS FORTRAN Application Programming: Language Reference, form GC26-3986, for further details.

The CLOSE Statement

The CLOSE statement may be used in VS FORTRAN to disconnect a file to a FORTRAN I/O unit. See the IBM publication, VS FORTRAN Application Programming: Language Reference, form GC26-3986, for further details.

The INQUIRE Statement

An INQUIRE statement may be used in VS FORTRAN to obtain information about an external file or FORTRAN I/O unit. See the IBM publication, VS FORTRAN Application Programming: Language Reference, form GC26-3986, for further details.

FORTRAN-II Statements

The FORTRAN-II statements accepted by FORTRAN-IV are:

```
READ  format,list
PRINT format,list
PUNCH format,list
```

The unit defaults for these statements are as follows:

```
READ  to SCARDS
PRINT to SPRINT
PUNCH to SPUNCH
```

VS FORTRAN implements the READ and PRINT statements which are contained in the FORTRAN 77 language standard. These are not considered to be "obsolete" FORTRAN-II statements. The PUNCH statement is not contained in the FORTRAN 77 and therefore is not implemented in VS FORTRAN.

October 1983

THE FORTRAN I/O COMMAND LANGUAGE MONITOR

The FORTRAN I/O Command Language Monitor serves as an interface between the user and the FORTRAN I/O routines. A predefined set of commands is available for use either under program control via the FTNCMD subroutine (see the section FTNCMD for the calling sequence), or conversationally after an error has occurred. The FORTRAN Monitor will intercept errors that occur during I/O operation and program interrupts.

The FORTRAN I/O Command Language Monitor exists so that programs may dynamically alter the I/O environment in which the program is to run. For example, a user may assign, default and equate logical I/O units from within his program, and he may choose some of the I/O options available with the SET command. The second purpose is to interrogate the Monitor for information which may help to explain why an error occurred. For example, one may wish to see the format that was currently being used, or one may wish to see a traceback of routines called when the error occurred. The FORTRAN Monitor will display this information, then coordinate error recovery so that the program can continue execution. The output from the FORTRAN monitor is written on the logical I/O unit SERCOM and the prompting input is read from the logical I/O unit GUSER.

FORTRAN I/O Commands

In the following list, the underlined characters indicate the minimum acceptable abbreviation of the command.

ASSIGN unit=FDname
BUFFER unit [LENGTH=integer]
CALL program-name
CLOSE unit
DEFAULT unit=FDname
DISPLAY keyword
EQUATE unit=unit
EXPLAIN [command name]
FTN
HELP
MODIFY keyword=value
MTS
QUERY
RELEASE unit
RETURN [DEFAULT|SKIPIO]
SET keyword=value
STOP
TRACEBACK

Any line beginning with a dollar sign is executed as an MTS command. Note that for most of the commands more than one set of parameters can

be specified on the same command. The equal sign (=) is optional in these commands.

The following list gives a detailed description of each command:

(1) ASSIGN unit=FDname

The ASSIGN command is used to assign or reassign any MTS logical I/O unit. This unit must be any one of 0,...,99, GUSER, SCARDS, SERCOM, SPRINT, or SPUNCH. For example, the command

```
ASSIGN 12=-FILE
```

assigns MTS I/O unit 12 to the file named -FILE.

(2) BUFFER unit [LENGTH=integer]

The BUFFER command defines a FORTRAN unit which is to be used exclusively for buffer-to-buffer I/O. For example,

```
BUFFER 9
```

defines FORTRAN unit 9 as a buffer-to-buffer I/O unit. Buffer-to-buffer I/O allows the user to write a formatted record, and then read it back (perhaps using a different format) without using a secondary storage device. "LENGTH=" specifies the length of the buffer, which cannot exceed 32767. LENGTH defaults to 256. "unit" must be one of 0-99. Buffer I/O is available only for a single record -- a user may read only one record, although that record may be read several times.

(3) CALL program-name

The call command is used to reinvoke any program in the backwards linkage chain, which is available using the TRACEBACK command. For example,

```
CALL MAIN
```

will rerun your program. Responsibility for providing serial reusability (such as repositioning I/O units to the beginning) is left to the programmer.

(4) CLOSE unit

The CLOSE command causes a unit at the FORTRAN level to be closed and its buffers released. The unit remains attached at the MTS level, but the next time it is used, the I/O library will reopen the unit and acquire new buffers. This command is primarily useful if the unit is attached to a magnetic tape which has several files with different blocking factors. The unit should be closed each time the tape is repositioned to a new file.

"unit" may be any legal FORTRAN unit from 0 to 99.

(5) DEFAULT unit=FDname

The DEFAULT command is used to default any MTS logical I/O unit that is not already assigned. This command has no effect if the unit is already assigned. Note that MTS automatically defaults SCARDS, SPRINT, GUSER, SERCOM, and SPUNCH (SPUNCH defaults only in batch mode if the CARDS global parameter is specified on the $SIGNON command). Note also that the FORTRAN I/O routines default units 5 and 6 to *SOURCE* and *SINK*, respectively; no other defaults are automatic. "unit" must be one of 0-99, GUSER, SCARDS, SERCOM, SPRINT, or SPUNCH.

(6) DISPLAY keyword

In conversational mode, the DISPLAY command is used to elicit information from the FORTRAN monitor. In the list of keywords that follows, the underlined characters indicate the minimum acceptable abbreviation of the command. Longer abbreviations will be accepted.

DEFAULT
FEEDBACK
FORMAT or FMT
FRi
FRS
GRi
GRS
LEVEL or LVL
LINE
MAP
MESSAGE or MSG
NAMELIST
NUMBER or NO
PSW
TRACEBACK
UNITS
YARDSTICK

(a) DISPLAY DEFAULT

This command is useful only after an error. The command will print the default corrective action associated with the error.

(b) DISPLAY FEEDBACK

This is useful only after an error. It will print diagnostic information concerning the error, such as the type of operation, the unit and file or device on which the error occurred, and the line or record number. The amount

of information printed is controlled by the SET FEEDBACK command.

(c) DISPLAY FORMAT
DISPLAY FMT

This is useful only after formatted I/O errors. It prints the format, either compiled or variable, that was being used at the time of the error.

(d) DISPLAY FRi

This is used after a program interrupt to display the contents of the floating-point register specified by i.

(e) DISPLAY FRS

This is used after a program interrupt to display the contents of all the floating-point registers.

(f) DISPLAY GRi

This is used after a program interrupt to display the contents of the general register specified by i.

(g) DISPLAY GRS

This is used after a program interrupt to display the contents of all the general registers.

(h) DISPLAY LEVEL
DISPLAY LVL

This is useful only after an error has occurred. It displays the error severity level.

(i) DISPLAY LINE

This is useful only after an I/O error in which a formatted line or a NAMELIST line was read. It will echo the input line read.

(j) DISPLAY MAP

This command produces a storage map showing the locations and the lengths of the user's currently loaded routines.

(k) DISPLAY MESSAGE
DISPLAY MSG

This is useful only after an error has occurred. It will repeat the error message.

(l) DISPLAY NAMELIST

This command causes the monitor to list the variable names in the namelist dictionary along with their types and dimensions. The NAMELIST keyword is valid only after a NAMELIST I/O error has occurred.

(m) DISPLAY NUMBER
DISPLAY NO

This is useful only after an error. The error number will be printed.

(n) DISPLAY PSW

This is used after a program interrupt to display the PSW (Program Status Word).

(o) DISPLAY TRACEBACK

This will print the subprogram linkage traceback.

(p) DISPLAY UNITS

This command will print a table showing the default files or devices for FORTRAN I/O units, and the unit assignments of other relevant files and devices.

(q) DISPLAY YARDSTICK

This prints several lines as a ruler with numbers to facilitate counting columns and character positions.

(7) EQUATE funit=munit

This command sets a FORTRAN unit equal to a MTS logical I/O unit. For example,

```
EQUATE 9=SERCOM
```

will cause any I/O operation that specifies unit 9 to be done on MTS logical unit SERCOM. "funit" must be one of 0-99, and "munit" must be one of 0-99, GUSER, SCARDS, SERCOM, SPRINT, or SPUNCH.

(8) EXPLAIN [command-name]

This command may be used to obtain information about the FORTRAN monitor command language.

(a) EXPLAIN

This will print a list of all the currently available FORTRAN monitor commands in their prototype form.

(b) EXPLAIN command-name

This will print a brief explanation of the FORTRAN monitor command whose name is given. For example,

```
EXPLAIN DISPLAY
```

explains the DISPLAY command. The command name abbreviations outlined above may be used.

(9) FTN

Normally when FTNCMD is called, the FORTRAN monitor will execute the given command and then return control to the program. However, if the FTN command is issued, the FORTRAN monitor will remain in control until the RETURN command is given by the user. In other situations, the FTN command is meaningless.

(10) HELP

This command should be issued if a novice FORTRAN user is having difficulty. It tells him how to get back to MTS command mode, and also how to obtain information about other FORTRAN monitor commands.

(11) MODIFY keyword=value

```
MODIFY LEVEL={0|1|2|3|4}
```

This command is useful only after an error has occurred. It is used to adjust the error severity level associated with the error. The error severity level can be determined by using the DISPLAY LEVEL command. Error severity levels have the following interpretations:

```
Level 0:  never print diagnostic, take default.
Level 1:  print only one diagnostic, take default.
Level 2:  print diagnostic, take default.
Level 3:  print diagnostic, query user about default.
Level 4:  print diagnostic, enter command mode.
```

In batch mode, the execution of the user's program is terminated whenever an error with the severity level greater than 2 is encountered.

```
MODIFY LEVEL={0|1|2|3|4} [FOR] n [ERRORS]
```

This form of the command may be used at any time to set the error severity level of a particular error or range of errors. n is the error number whose severity level is to be changed. The error number is printed in batch debug output after an error, or is available with the DISPLAY NUMBER command after an I/O error has occurred in conversational mode. n may also be a list of error numbers separated by commas or a range of errors of form low-high where the high must be greater than low.

```
MODIFY LEVEL=1 FOR 211
MODIFY LEVEL=2 223,233
MODIFY LEVEL=0 FOR 208-211 ERRORS
```

(12) MTS

The MTS command returns control to MTS command mode. The FORTRAN monitor may be reentered by issuing the $RESTART command.

(13) QUERY keyword

This command returns information to a program variable or array. The amount and type of information returned depend on the keyword. The information is returned in the last parameter supplied in the call to FTNCMD. For example,

```
CALL FTNCMD('QUERY WARN;',0,L)
```

returns information in the logical*4 variable L. Note: VS FORTRAN programs should normally use the INQUIRE statement.

(a) QUERY ERR [FOR {CATEGORY|ERROR} n]

This returns .TRUE. or .FALSE. to a logical*4 variable according to whether ERR has been set ON or OFF with the SET command for all errors, for the whole of the specified category of errors, or for the single specified error. If no SET ERR command was issued, the default is returned.

(b) QUERY {ATTENTION|ATTN}
QUERY AUTODBL
QUERY ERRMSG
QUERY MINUSZERO
QUERY MODECHECK
QUERY NAMEOUT
QUERY {NULLBLANK|NULLBLNK}
QUERY QUIT
QUERY SEMIFREE
QUERY UVCHECK
QUERY WARN
QUERY WRAPAROUND
QUERY ZEROSUPPRESS

This returns .TRUE. or .FALSE. to a logical*4 variable according to the current setting of the given keyword. Each of them has a default setting, or it could have been set ON (.TRUE.) or OFF (.FALSE.) with the SET command. See the SET command for details of these keywords.

(c) QUERY FEEDBACK

Returns the value to INTEGER*4 variable as follows: 0 if FEEDBACK is NONE, 1 if POOR, 2 if NORMAL, 3 if GOOD. FEEDBACK is the amount of detail given on what was going on when an error occurred.

(d) QUERY NAMEFMT

Returns the characters "FREE" or "IBM " to a fullword variable, whichever is the current setting for the format used in NAMELIST I/O.

(e) QUERY {PREFIX|PFX}

Returns to the first byte of a variable of any type, the current FORTRAN Monitor prefix character. This character can be changed with the SET command.

(f) QUERY ORL [[FOR [UNIT]] n]

Returns to an INTEGER*4 variable the current output record length for FORTRAN Monitor output or for output to the specified unit. This length can be changed by the SET command.

(g) QUERY {LEVEL|LVL} [[FOR] n]

Returns the severity level for the last error or for the specified error to an INTEGER*4 variable. The level can be set with the MODIFY command. If no error number is specified and there is no error yet, -1 is returned. The possible levels are from 0 (the least severe) to 4 (the most severe).

(h) QUERY INDEXED n
QUERY SEQUENTIAL n
QUERY FORMATTED n
QUERY UNFORMATTED n
QUERY BUFFERED n
QUERY DEFINED n

Returns .TRUE. or .FALSE. to a logical*4 variable according to whether or not the specified FORTRAN unit has such a property. INDEXED and SEQUENTIAL, and FORMATTED and UNFORMATTED are pairs of opposites, one of which will always be true and the other false unless the unit n is not in use

and hence has no properties at all. BUFFERED refers to the use of unit n as a buffer-to-buffer unit (see the BUFFER command), and DEFINED to its use as a DEFINE-FILE unit.

(i) QUERY DEFTYPE n

If unit n is a define-file unit, this command returns a single character "U", "E" or "L" to the first byte of a variable of any type to give the DEFINE-FILE type. If n is not in use or is not a define-file unit, a blank is returned.

(j) QUERY {RECORDCOUNT|RECCNT} n
QUERY {LINENUMBER|LNR} n
QUERY LENGTH n
QUERY {RETURNCODE|IORC|RC} n

Returns the current count of records processed, MTS line number (1.000 is returned as 1000), last record length, or last return code (0 is normal) for the specified unit n to an INTEGER*4 variable. If unit n is not in use, zero is returned.

(k) QUERY OPERATION [n]

Returns the number of the last I/O operation to be performed (or the last operation on unit n) to an INTEGER*4 variable. Possible numbers are: 0=Initialize, 1=Formatted READ, 2=Formatted WRITE, 3=Unformatted READ, 4=Unformatted WRITE, 5=FIND, 6=BACKSPACE, 7=REWIND, 8=ENDFILE, 9=STOP, 10=PAUSE, 11=CALL EXIT, 12=Error Monitor call, 13=NAMELIST READ, 14=NAMELIST WRITE, 15=DEFINE FILE, 16=DEBUG, 17=EMPTYF, 18=list-directed READ or 19=list-directed WRITE. This list is subject to change.

(l) QUERY {DEVICETYPE|DEVTYP} n

Returns the device type of the unit n to an INTEGER*4 variable. If the unit n is not in use, zero is returned. The possible values are 1=line file, 2=sequential file, 3=magnetic tape, 4=*DUMMY*, 5=no device allocated, 6=unit record (such as a card reader or miscellaneous). This list is subject to change.

(m) QUERY {ERRORNO|IOERRNO|ERRNO} [n]

Returns the error number of the last I/O error to occur or the last error on the unit n to an INTEGER*4 variable. If there is no error, zero is returned.

(n) QUERY {IOERRCAT|ERRCAT} [n]

Returns the category number of the last I/O error to occur or of the specified error to an INTEGER*4 variable. If no error number is specified and there has not yet been an error, 0 is returned. Possible category numbers are from 1 to 9. For details, see Appendix A to this section.

(o) QUERY ASSIGN n
QUERY DEFAULT n

Returns .TRUE. or .FALSE. to a logical variable according to whether or not the unit n has been set via an ASSIGN or DEFAULT command, respectively. .FALSE. is returned if unit n is not in use. .TRUE. is returned in the DEFAULT case if unit n was defaulted automatically by the FORTRAN Monitor. For DEFAULT, "n" must be between 0 and 99.

(p) QUERY FDNAME n

Returns a character string as the current FDname (postfixed with a blank) of the specified FORTRAN unit n to an array of any type. Examples of FDnames are *SOURCE*, *T*, MYFILE. The array must be made large enough to hold the longest possible FDname; otherwise, the array could be overflowed with unpredictable results. 60 characters or 15 fullwords give enough space for any FDname encountered. If unit n is not in use, a single blank will be returned.

(q) QUERY EQUATE n

Returns an eight-character string as the unit name (e.g., "GUSER ") or number (e.g., "3 ") to which unit n is currently equated by an EQUATE command. If the unit n is not in use or was not equated, a single blank will be returned. Note that unit numbers are returned in character form, left-justified with trailing blanks, and not in numeric form.

(14) RELEASE unit

This command will close and release buffers of a unit at the MTS level. The unit remains attached at its current position. This command may be used to release buffer space that is not required for a certain period of time.

"unit" may be any legal FORTRAN unit from 0 to 99.

Note: buffer space is automatically acquired for each unit that is being used by the FORTRAN program. The RELEASE command releases this buffer space. It will also release the buffer space that is acquired with the BUFFER command.

(15) RETURN [DEFAULT|SKIPIO]

The RETURN command causes the FORTRAN monitor to return control to the user program. It has three forms:

(a) RETURN

The RETURN command is used to return to the user program after calls to FTNCMD, and to return to MTS after program interrupts. For I/O errors, RETURN is treated the same as RETURN DEFAULT.

(b) RETURN DEFAULT

This is useful after all types of errors, and causes the FORTRAN monitor to take the default corrective action associated with the error.

(c) RETURN SKIPIO

This is useful after an I/O error. It will cause the FORTRAN monitor to ignore the remainder of the I/O operation and resume execution at the next statement in the program.

(16) SET keyword=value

The SET command is used to control the behavior of the FORTRAN monitor. The following keywords and values are defined for the SET command. Minimum abbreviations (when available) are underlined.

{ATTENTION|ATTN}={ON|OFF}
AUTODBL={ON|OFF}
{CARRIAGECONTROL|CC}={ON|OFF|MACHINE|DEFAULT} FOR UNIT n
ERR={ON|OFF} [FOR {CATEGORY|ERROR} n]
ERRMSG={ON|OFF}
FEEDBACK={NONE|POOR|NORMAL|GOOD}
{LOWERCASE|LC}={ON|OFF} FOR UNIT n
MCC={ON|OFF} FOR UNIT n
MINUSZERO={ON|OFF}
MODECHECK ={ON|OFF}
NAMEFMT={IBM|FREE}
NAMEOUT={ON|OFF}
{NULLBLANK|NULLBLNK}={ON|OFF}
ORL=n
{PREFIX|PFX}=character
QUIT={ON|OFF}
{UPPERCASE|UC}={ON|OFF} FOR UNIT n
UVCHECK={ON|OFF}
WARN={ON|OFF}
WRAPAROUND={ON|OFF}
ZEROSUPPRESS ={ON|OFF}

(a) SET ATTENTION={ON|OFF}
SET ATTN={ON|OFF}

If ATTN is set to ON, an attention interrupt during execution of the monitor will be intercepted and control will be returned to the user in command mode. If ATTN is OFF, an attention interrupt in the monitor will be handled either by the user's attention interrupt trap (if any), or by MTS. The default is ON.

(b) SET AUTODBL={ON|OFF}

If ON, this sets the "auto double" feature in FORTRAN-H Extended. The default is OFF. This has no effect in ordinary FORTRAN I/O.

(c) SET CARRIAGECONTROL={ON|OFF|MACHINE|DEFAULT} FOR UNIT n
SET CC={ON|OFF|MACHINE|DEFAULT} FOR UNIT n

If CARRIAGECONTROL is set to ON, the MTS @CC FDname modifier is applied. If set to OFF, the @¬CC FDname modifier is applied. If set to MACHINE, the MTS @MCC FDname modifier is applied. If set to DEFAULT, no modifier is applied. The default is dependent on the setting of the MTS @CC and @MCC FDname modifiers for the unit.

(d) SET ERR={ON|OFF}

If ERR is set to ON, and an I/O error occurs, the FORTRAN monitor will take the ERR exit provided on the READ or WRITE statement. The default is ON.

SET ERR={ON|OFF} FOR {CATEGORY|ERROR} n

Here only just one category of errors or one individual error is affected.

(e) SET ERRMSG={ON|OFF}

If ERRMSG is set to ON, the FORTRAN monitor will print a diagnostic message before it takes the ERR exit on a FORTRAN READ statement. ERRMSG has no effect when ERR=OFF. The default is ON.

(f) SET FEEDBACK={NONE|POOR|NORMAL|GOOD}

This keyword controls the amount of diagnostic information produced by the FORTRAN monitor after an I/O error. In conversational mode, the default is POOR if the MTS command $SET TERSE=ON is in effect, and NORMAL if $SET TERSE=OFF is in effect (the default). In batch mode, the default is GOOD, independent of the TERSE setting.

(g) SET LOWERCASE={ON|OFF} FOR UNIT n
SET LC={ON|OFF} FOR UNIT n

If LOWERCASE is set to ON, the READ and WRITE statements will not convert input/output to uppercase on FORTRAN unit "n". If set to OFF, this conversion is not performed. The default is dependent on the MTS @LC and @UC FDname modifiers for the unit.

(h) SET MCC={ON|OFF} FOR UNIT n

If MCC is set to ON, the first character of each line is treated as a machine carriage-control character for FORTRAN unit "n". If set to OFF, machine carriage-control is not assumed. The default is OFF.

(i) SET MINUSZERO={ON|OFF}

If MINUSZERO is set to ON, a blank field on a formatted READ of REAL numbers will return -0 rather than +0. This can be used to distinguish a field which is blank (missing data) from a field which contains a true zero. The default is OFF, i.e., blank fields are returned as +0.

(j) SET MODECHECK={ON|OFF}

If MODECHECK is set to ON, the variable type must agree with the format code. If MODECHECK is set to OFF, any format code can be specified, regardless of the type. The variable will be converted according to the format code. The default is ON.

(k) SET NAMEFMT={IBM|FREE}

If NAMEFMT is set to IBM, NAMELIST input must be in the standard IBM OS NAMELIST format. The first record should start with "&naml" in column 2, where &naml is the name of the namelist. The last record should end with "&END" in column 2. If NAMEFMT is set to FREE, free-format NAMELIST input will be recognized. The default is FREE.

(l) SET NAMEOUT={ON|OFF}

If NAMEOUT is set to ON, then &naml and &END are not printed on the first and last lines for NAMELIST output. The default is OFF.

(m) SET NULLBLANK={ON|OFF}
SET NULLBLNK={ON|OFF}

If NULLBLANK is set to ON, a null line on output will result in a single blank. This is useful if the output is

being put in a file which will ultimately be copied to a printer. The default is OFF.

(n) SET ORL=m

This keyword controls the length of all output lines written by the FORTRAN monitor (excluding lines printed by the EXPLAIN command). "m" may range from 20 to 128. The default is the output record length of *MSINK*.

SET ORL=m FOR [UNIT] n

This command controls the length of all output lines written to FORTRAN unit "n". "m" may range from 20 to 32767. The default is the maximum output record length supplied by the MTS system.

(o) SET PREFIX=character
SET PFX=character

This keyword is used to change the prefix character used by the FORTRAN monitor. The default prefix character is the at sign (@). Only one character is allowed.

(p) SET QUIT={ON|OFF}

If QUIT is set to ON, then an I/O error in batch mode will result in the job being terminated (signed off). QUIT has no effect in conversational mode. The default is OFF.

(q) SET SEMIFREE={ON|OFF}

If SEMIFREE is set to ON, semifree input is allowed. If set to OFF, standard format conventions will be followed. The default is ON.

(r) SET UPPERCASE={ON|OFF} FOR UNIT n
SET UC={ON|OFF} FOR UNIT n

If UPPERCASE is set to ON, the READ and WRITE statements convert input/output to uppercase on FORTRAN unit "n". If set to OFF, this conversion is not performed. The default is dependent on the settings of the MTS @LC and @UC FDname modifiers for the unit.

(s) SET UVCHECK={ON|OFF}

If UVCHECK is set to ON (the default), any attempt to print a numeric variable whose value is identical to a core constant (X'81818181') will produce a warning. The output field will be filled with the character "U". If UVCHECK is set to OFF, this checking is not performed.

(t) SET WARN={ON|OFF}

If WARN is ON, warning messages are printed on *MSINK*. If WARN is OFF, warning messages are suppressed. This is useful if, for example, the user intentionally prints a number too large for the output field in order to fill the field with asterisks (*). The default is ON. Note: The definition of a warning message versus an error message is built into the I/O library and cannot be changed by the user.

(u) SET WRAPAROUND={ON|OFF}

If WRAPAROUND is set to ON, any output lines longer than the device output record length will be continued on the next line of the device. If set to OFF, any line longer than the device output record length will be truncated to that length. The default is ON.

(v) SET ZEROSUPPRESS={ON|OFF}

This keyword is used to suppress printing of true zeros on output. If ZEROSUPPRESS is set to ON, the field will be left blank if an integer or floating-point variable is equal to zero. If ZEROSUPPRESS is set to OFF, zeros will be printed in the field. The default is OFF.

(17) STOP

The STOP command terminates execution, like the FORTRAN STOP statement. The program may not be restarted.

(18) TRACEBACK

This command prints a subprogram linkage traceback.

The FTNCMD Subroutine

The FTNCMD subroutine may be called to execute a FORTRAN I/O command, or an MTS command, in a program and to return to the program after the command has been executed.

FTNCMD is an entry point in the FORTRAN I/O command language monitor, and thus, is available automatically with the I/O library.

The subroutine FTNCMD can be called from a FORTRAN program as follows:

```
CALL FTNCMD(char[,len])
```

where:

char — is the character string containing a FORTRAN I/O command or an MTS command. If the character string is terminated by a semicolon, the length parameter len may be omitted. It is possible to build the command substituting variables in the character string.

len — is the length of the command character string expressed as either a fullword or a halfword integer (INTEGER*4 or INTEGER*2). This parameter may be omitted if the command character string is terminated with a semicolon (;).

If the first character in the call to FTNCMD is a dollar sign ($), the subroutine returns control to MTS. The character string is then executed as an MTS command. After execution, MTS returns control to the program.

If the first character in the command is not a dollar sign, the subroutine assumes it is a FORTRAN I/O command rather than an MTS command. The FORTRAN command language monitor executes the I/O command and returns to the program.

Text replacement is possible by using a question mark (?) in the command string at the point where text is to be substituted and providing the replacement text as an additional parameter. In this case, replacement parameters begin at the third parameter position. The following commands are identical.

```
(1)  CALL FTNCMD('ASSIGN 10=-S',12)

(2)  CALL FTNCMD('ASSIGN 10=-S;')

(3)  J=10
     CALL FTNCMD('ASSIGN ?=-S;',0,J)
```

Note that if a variable such as J in the last example, the length argument (the second argument in the FTNCMD call) must exist but can be set to zero if a semicolon is used to terminate the string.

The substitution of the variable argument may involve conversion. If the argument is a character string, the characters terminated by the first blank are substituted directly. The terminating blank is not substituted. If the variable is an integer value, it is converted to a character string before substitution. The following are equivalent:

```
(1)  LOGICAL*1 NAME(3) /'-','S',' '/
     CALL FTNCMD('ASSIGN 10=?;',0,NAME)

(2)  J=10
     CALL FTNCMD('ASSIGN ?=-S;',0,J)
```

The following examples illustrate the use of FTNCMD.

(1) The following command assigns MTS unit 4 to the temporary file -SCRATCH4 from within the program.

```
CALL FTNCMD('ASSIGN 4=-SCRATCH4',18)
```

or

```
CALL FTNCMD('ASSIGN 4=-SCRATCH4;')
```

(2) The following command specifies that FORTRAN unit 9 is to be used for core-to-core I/O with a buffer length of 300.

```
CALL FTNCMD('BUFFER 9 LENGTH=300;')
```

(3) The following command defaults MTS unit SPUNCH to *PUNCH*.

```
CALL FTNCMD('DEFAULT SPUNCH=*PUNCH*;')
```

Note: This is only effective if the unit SPUNCH has not already been assigned. Note also that SPUNCH would automatically default to *PUNCH* in batch mode if the CARDS global parameter was specified on the $SIGNON command.

(4) The following command maps FORTRAN unit 99 to MTS unit SERCOM.

```
CALL FTNCMD('EQUATE 99=SERCOM;')
```

(5) The following command gives control to the FORTRAN I/O command language monitor.

```
CALL FTNCMD('FTN;')
```

Any of the FORTRAN I/O commands except the QUERY command may be issued by the user at this point. Return is made to the calling program when the RETURN command is issued.

(6) Many statistical applications require that missing data be distinguished from data with a value of zero. If a blank record is read under I, F, D, E, or G format, the variables in the I/O list are set to zero. A blank field for a REAL or COMPLEX variable will be set to -0.0 if the following statement is executed before the read operation:

```
CALL FTNCMD('SET MINUSZERO=ON;')
```

(7) The following command enters debug mode to debug a program from a terminal.

```
CALL FTNCMD('$SDS;')
```

Return is made automatically when the CONTINUE debug command is given.

Error processing

The FORTRAN I/O Command Language Monitor controls error processing and monitors error recovery in conversational mode. All errors and warnings are produced on *MSINK*, the printer in batch mode or the terminal in conversational mode. The messages may not be reassigned to a different device, although they may be suppressed or the severity level altered such that the error message will occur only once and the default corrective action will be automatically taken. See the MODIFY command description in the section "FORTRAN I/O Commands."

(1) Batch Mode

If an error occurs in batch mode a predefined set of diagnostics is given and execution is terminated. The following is an example of a batch error message that occurred when invalid data was entered on the READ request.

```
######################################################################
######################################################################

***ERROR*** Invalid integer field, illegal character. Condition
            occurred during a formatted read on FORTRAN Unit 5
            which is attached to *SOURCE*. The read was sequen-
            tial at record number 1. A return will be made to
            the system.

This ruler indicates column numbers in the data line which
follows; if a dollar sign appears below the line it indicates
the probable location of the error:

         10        20        30        40        50        60
....|....|....|....|....|....|....|....|....|....|....|....|....
1,2,4.5,5.6,3,
     $

######################################################################
######################################################################

Batch FORTRAN debug output follows,

Format-- (3I5,2F5.2,I5)

Subprogram traceback,
Error occurred in routine MAIN at hexadecimal displacement +12C
Called from "SYSTEM"

Unit usage table
Unit 5 is assigned to *SOURCE*
Unit 6 is assigned to *SINK*
Guser is assigned to *MSOURCE*
Scards is assigned to *MSOURCE*
```

```
Sercom is assigned to *MSINK*
Sprint is assigned to *MSINK*

Error number=211

Severity level=2

FORTRAN error causes a return to the system.

Execution terminated
```

From the format printed in the above example, it is quite obvious that the data line should have an INTEGER value as the the third data value not a REAL value.

(2) Conversational Mode

In conversational mode if the error occurs on data entry (i.e., invalid data as in the example above) the user is prompted to reenter the line from the start of the field in error. The line that is to be reentered is first echoed on the terminal. At this point, the error should be corrected and the line re-entered. If a null line is entered, remaining items in the I/O list will be set to zero or blank. If "CANCEL" is entered, the user is placed in command mode. The message:

```
@@@ FORTRAN @@@
```

is written when the FORTRAN I/O Command Language Monitor is invoked. The prefix character will be an "@" if the FORTRAN Monitor is in control.

In command mode the user may use any of the FORTRAN I/O commands listed in the preceding section to display values and gain more information of the error, and to take some recovery action.

If an error occurs which is not an error on data entry, the error message and default action are printed on the terminal. For example, the error was an actual I/O error. The user is then prompted whether or not he wants to take the default action. If he does not want the default, he can enter FORTRAN I/O Command Language Monitor mode and use the commands (such as "RETURN SKIPIO") listed in the preceding section to gain some more information or take some recovery action.

The following examples in conversational mode illustrate some of the alternatives mentioned above. The user's input is shown in lowercase, the prefix character "@" is used when the FORTRAN I/O Command Language Monitor has control.

Example 1:

```
# $run -load
# Execution begins
   1,2,4.5,5.6,3,

  ***ERROR*** Invalid integer field, illegal character.  Condi-
              tion  occurred  during a formatted read on FORTRAN
              Unit 5 which is  attached  to  *SOURCE*.  If  you
              re-enter  the  rest  of the line, execution of the
              input statement will continue.
   1,2,4.5,5.6,3,
         $
  Re-enter line from beginning of field in  error  or  "CANCEL".
  Rest of line now is:
  4.5,5.6,3,
  3,4.5,5.6,3,

     1    2    3 4.50 5.60    3
# Execution terminated
```

Example 2:

This example is the same as above except that when prompted to reenter the line the user cannot correct the error because he does not remember the format that was currently being used. He, therefore, enters command mode to display the format. The conversation is shown from the reenter request.

```
  Re-enter line from beginning of field in  error  or  "CANCEL".
  Rest of line now is:
  4.5,5.6,3,
  cancel
  @@@ FORTRAN MONITOR @@@
@ dis format
@ FORMAT--(3I5,2F5.2,I5)
@ return
  Re-enter  line  from  beginning of field in error or "CANCEL".
  Rest of line now is:
  4.5,5.6,3,
  3,4.5,5.6,3,

     1    2    3 4.50 5.60    3
# Execution terminated
```

October 1983

APPENDIX A: FORTRAN I/O LIBRARY ERROR MESSAGES

The FORTRAN I/O Library error messages are divided into nine categories:

(1) Variable-format decoder errors
(2) Inappropriate calls for I/O operation
(3) Errors during I/O operations
(4) Define file errors
(5) Format errors
(6) NAMELIST errors
(7) List-directed I/O errors
(8) Reserved
(9) Miscellaneous errors

Each message is preceded by two numbers: its error number and its severity level. The severity levels have the following interpretations:

0: Never print diagnostic, take default.
1: Print only one diagnostic, take default.
2: Print diagnostic, take default.
3: Print diagnostic, query user about the default.
4: Print diagnostic, enter command mode.

Variable-Format Decoder Errors

1-1 Variable formats must be enclosed in parentheses.

The first character of the format is not a left parenthesis. This is a warning.

Default: The error will be ignored.

2-1 The maximum repeat count is 255.

This is a warning.

Default: Count=255.

3-3 Invalid variable format, unexpected integer.

An integer precedes a T-format code, quote, comma, slash, or right parenthesis.

Default: The integer will be ignored.

4-3 Invalid variable format, unexpected character or missing closing parenthesis.

There was an invalid character, illegal character in format field, or a misplaced character, number or period.

Default: The format will be assumed terminated by the unexpected character.

5-3 FORTRAN I/O failure detected by variable format decoding routine.

This is a system error.

6-1 Invalid variable format, missing repeat count.

A number does not precede an X-format code. This is a warning.

Default: Count=1.

7-1 Invalid variable format, null quote string.

This is a warning.

Default: The null quote string will be ignored.

9-3 Invalid variable format, unpaired right parentheses.

Default: The format will be assumed terminated by the unexpected right parenthesis.

10-1 Invalid variable format, missing specification.

Two successive commas or a comma and a right parenthesis were encountered. This is a warning.

Default: The error will be ignored.

11-1 Invalid variable format; comma, colon or slash should separate format fields.

A comma or a slash was expected. This is a warning.

Default: A comma will be assumed, and the error will be ignored.

16-3 Invalid variable format, field width > 255.

Default: The format will be assumed terminated at the point of the error.

17-1 Invalid variable format, missing column number.

A nonnumeric character follows a T-format code. This is a warning.

Default: Column=1.

18-3 Invalid variable format, missing Hollerith count.

A number does not precede a H-format code.

Default: The format will be assumed terminated at the point of the error.

19-1 Invalid variable format, column number > 32767.

This is a warning.

Default: Column=32767.

20-1 Invalid variable format, the absolute value of the scale factor > 127.

This is a warning.

Default: A scale factor of zero will be assumed.

21-3 Invalid variable format, missing closing quote.

None of 255 characters following the first quote is a quote.

Default: The format will be assumed terminated by the first quote.

22-3 Invalid variable format, Hollerith count > 255.

Default: The format will be assumed terminated at the point of the error.

Inappropriate Calls for I/O Operations

131-3 Unit number n was specified; FORTRAN units must be 0,...,99 only.

Default: The I/O statement will be ignored, and execution of the program will continue with the next statement.

132-3 Unit was referenced but was not assigned or defaulted.

Default: Presuming an assignment or reassignment of the unit, the I/O statement will be retried.

137-3 Illegal I/O device referenced. Attempt to msg.

"msg" may be one of the following:

write to an input device
read from a file without "read" access
write to a file without "write" access
read from an output device

Default: Presuming an assignment or reassignment of the unit, the statement will be retried.

138-3 Unit cannot be rewound.

Default: The REWIND request will be ignored.

139-3 Unit cannot be backspaced.

Default: The BACKSPACE request will be ignored.

140-3 The assigned file or device is non-exsistent or access is not allowed.

Default: Presuming an assignment or reassignment of the unit, the I/O statement will be retried.

141-2 Unit cannot be endfiled.

This is a warning.

Default: The ENDFILE request will be ignored.

143-3 Attempting to backspace an unformatted record on a unit record device.

Default: The BACKSPACE request will be ignored.

144-3 Attempting to do an indexed operation on a sequential file or device.

Default: The operation will be done sequentially if the unit is not reassigned. Otherwise, the I/O statement will be retried.

146-1 Endfiling a unit after a FIND.

This is a warning.

Default: Endfile is a null operation on files/devices (other than magnetic tapes).

147-1 Backspacing a unit after a FIND or a REWIND.

This is a warning.

Default: The unit will be backspaced.

148-3 Attempting to backspace before the beginning of a file.

Default: The BACKSPACE request will be ignored.

152-4 Unit number n is being referenced but was not defined.

Default: Presuming an assignment or reassignment of the unit, the I/O statement will be retried.

161-4 Attempt to read a formatted record with an unformatted READ statement.

Default: Control will be returned to the system.

162-4 The record read was probably a blocked record.

Currently, unformatted I/O routines cannot read VBS-formatted records.

Default: Control will be returned to the system.

449-2 Unrecognizable value for XXX parameter in YYY statement.

Default: Control will be returned to the system.

450-2 Invalid value of XXX for unit number parameter in YYY statement.

Default: Control will be returned to the system.

451-2 File name in INQUIRE statement invalid or too long, or file nonexistent, or fdub can't be obtained for file.

Default: Control will be returned to the system.

452-2 Request to OPEN unit n for direct access illegal - device cannot handle it.

Default: Control will be returned to the system.

453-2 STATUS=SSS specified for unit n in OPEN statement, but msg.

Default: Control will be returned to the system.

454-2 Return code of nn from attempt to create NEW file for OPEN statement.

Default: Control will be returned to the system.

455-2 Return code of nn from attempt to DELETE file for CLOSE statement.

Unable to destroy the file.

Default: Control will be returned to the system.

513-2 Illegal dimension statement encountered during I/O. Either lower bound greater than upper bound or value out of range.

Default: Control will be returned to the system.

514-2 Asynchronous READ statement not supported in MTS.

Default: Control will be returned to the system.

515-2 Asynchronous WRITE statement not supported in MTS.

Default: Control will be returned to the system.

516-2 Asynchronous WAIT statement not supported in MTS.

Default: Control will be returned to the system.

517-2 Asynchronous REWIND, BACKSPACE, or ENDFILE not supported in MTS.

Default: Control will be returned to the system.

Errors During I/O Operations

65-3 Unformatted input record too short, list length exceeds record length.

Default: The remainder of the READ will be skipped, and execution of the program will continue with the next statement.

66-3 Attempting to start an unformatted read in the middle of a spanned record. You probably have scrambled data.

Default: The READ statement will be ignored, and execution of the program will continue with the next statement.

67-4 Illegal subscript occurred during I/O operation.

Default: A return will be made to the system.

129-3 Unformatted input record too short. The length indicated in the block control word exceeds the length of the record.

Default: The READ statement will be ignored, and execution of the program will continue with the next statement.

130-3 End-of-file read, but no END= exit provided.

Default: Presuming an assignment or reassignment of the unit, the READ statement will be retried.

133-3 No buffer space is available for unit n.

This is a system error.

134-3 FORTRAN I/O failure detected by unit control routine; could not free space allotted to unit n.

This is a system error.

142-3 FORTRAN I/O failure detected by unit control routine, error return from CONTROL.

This is a system error.

145-4 Error return from READ or WRITE: msg

"msg" indicates the actual I/O error message.

Default: The rest of the current I/O operation is ignored. If the unit is reassigned any further I/O is performed on the new unit.

149-4 Error return from READ or WRITE: msg

"msg" indicates the actual I/O error message.

Default: The rest of the current I/O operation is ignored. If the unit is reassigned, any further I/O is performed on the new unit.

150-3 FORTRAN I/O failure detected by unit control routine, NOTE error.

This is a system error.

151-3 FORTRAN I/O failure detected by unit control routine, POINT error.

This is a system error.

Default: The I/O statement will be ignored.

168-4 A wait-to-lock was interrupted or locking would cause deadlock.

The MTS routines NOTE and POINT indicated the message.

Default: The I/O statement will be retried.

169-4 A Software/Hardware error occurred while attempting to open the file.

Default: Control will be returned to the system.

170-3 End-of-file during an internal READ.

171-3 Attempt to write more records than allowed during internal WRITE. Increase the number of elements in the array.

401-4 Return code of nn received from I/O device but no associated error message returned.

Define File Errors

153-3 ***DEFINE FILE*** - output record greater than specified in the DEFINE FILE statement.

Default: The record, truncated to the correct size, will be written.

154-3 ***DEFINE FILE*** - associated variable is < 0 which is illegal

Default: The rest of the I/O statement will be ignored.

155-4 ***DEFINE FILE*** - unit must be assigned to a sequential file.

Default: Presuming an assignment or reassignment of the unit, execution will continue.

156-4 ***DEFINE FILE*** - input file not formatted the same as when it was created.

Default: Assuming an emptying or reassignment of the unit, the statement will be repeated.

157-3 ***DEFINE FILE*** - index variable > specified maximum.

Default: The remainder of the I/O operation will be skipped, and execution will resume within the program at the next statement.

158-4 ***DEFINE FILE*** - format type different than that specified.

Default: The format will be ignored and execution will continue.

159-3 DEFINE FILE - non-indexed I/O statements are not allowed on "DEFINE FILE" data files.

Default: The offending I/O statement will be ignored.

160-2 DEFINE FILE - A second DEFINE FILE statement for the same unit has been encountered.

This is a warning.

Default: The redefinition will be ignored and execution continues with the next statement.

163-4 Unit number n was specified in a DEFINE FILE statement; FORTRAN units must be 0,...,99 only.

Default: A return will be made to the system.

164-4 DEFINE FILE could not initialize the file - cause - no write access or no file space

Default: Control will be returned to the system.

165-4 Illegal I/O device referenced. Attempt to msg.

This is same as the error message 137, except that I/O device is used in DEFINE FILE. "msg" is one of the following:

```
write to an input device
read from a file without "read" access
write to a file without "write" access
read from an output device
```

Default: Control will be returned to the system.

166-4 DEFINE FILE - concatenated files are not allowed when using DEFINE FILE.

Default: Control will be returned to the system.

167-4 DEFINE FILE - file contains bad data - e.g. variable length lines.

Default: Control will be returned to the system.

Format Errors

193-1 A format width, literal string, or column specification exceeds the device length.

This is a warning.

Default: This field cannot be wrapped so it will be truncated. Wraparound is still in effect for the remainder of the output list.

194-3 Invalid compiled format. Your program has most likely exceeded the dimensions of an array.

Default: The remainder of the I/O operation will be skipped, and execution will resume within the program at the next statement.

195-1 Attempting to write a formatted record too long for file or device.

This is a warning.

Default: The record will be truncated before being written.

196-1 Attempting to read a formatted record longer than file or device record length.

This is a warning.

Default: The record will be padded with blanks.

197-1 Variable (or array element) in output list is undefined.

This is a warning.

Default: A field of U's will be written.

198-3 Format code and variable type do not match. Format code is fmt, type is type.

"type" may be one of INTEGER, REAL, LOGICAL, COMPLEX.

Default: "G" format code will be used.

199-3 Illegal parameter encountered in call to SIOC.

FORMAT routines have passed an illegal parameter to SIOC routine.

Default: The rest of the I/O operation will be ignored.

200-1 Formatted record too long for file or device.

This is a warning.

Default: Wraparound will take effect.

201-3 fmt format can only be used if the variable is m or n bytes long.

If the MODECHECK is set to ON, the variable type must agree with the format code.

Default: "G" format code will be used.

202-3 fmt format can only be used if the variable is n bytes long.

Default: The rest of the I/O operation will be ignored.

206-1 Output field width too small.

This is a warning.

Default: A field of *'s will be written.

208-2 Invalid {INTEGER|REAL} number, bad syntax.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

211-2 Invalid type field, illegal character.

"type" may be INTEGER, REAL, LOGICAL, HEXADECIMAL. If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

213-2 Magnitude of REAL number exceeds machine limits:
(.539760534693402789D-78, .723700557733226211D+76)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

214-2 INTEGER*4 number exceeds machine limits:
(-2147483648, +2147483647)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

215-2 INTEGER*2 number exceeds machine limits: (-32768, +32767)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

NAMELIST Errors

257-3 Variable subscripts are illegal in NAMELIST input.

For example, "ABC(XYZ)" was specified instead of "ABC(15)".

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

258-2 Variable name appears subscripted but was not dimensioned.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

259-2 Variable name could not be found in NAMELIST dictionary.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

260-2 Illegal character encountered in NAMELIST input.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

261-3 Expecting a data value, but end of record found.

An unexpected end of data was encountered in NAMELIST read with free-format.

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

262-3 Expecting a closing quote, but end of record found.

An end of record was encountered during the process of a literal string for both IBM and free-formats.

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

263-3 Expecting a closing parentheses, but end of record found.

An end of record was encountered when NAMELIST routines were processing a variable name with subscripts.

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

264-2 Expecting imaginary part of COMPLEX number.

An illegal character was encountered. If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

265-2 Subscript exceeds corresponding array dimension.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

266-2 Repetition factor exceeds array (or element) size.

Default: Extra values will be ignored and the read continued.

267-2 Name > 6 characters.

A variable name in the NAMELIST is too long. If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

268-2 Missing bracket ')' in COMPLEX input.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

269-2 Invalid type number, msg.

"type" may be one of REAL, INTEGER, LOGICAL, HEXADECIMAL; and "msg" may be "illegal character" or "bad syntax". If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

273-2 Magnitude of REAL number exceeds machine limits: (.53976053469340278 9D-78, .72370055773322621 1D+76)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

274-2 INTEGER*4 number exceeds machine limits: (-2147483648, 2147483647)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

275-2 INTEGER*2 number exceeds machine limits: (-32768, 32767)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

276-1 Variable (or array element) in output list is undefined.

This is a warning.

Default: A field of U's will be written.

277-3 Output field width too small.

Default: A field of *'s will be written.

278-3 Repetition factors (denoted by the '*') are illegal in this context.

The repetition factors cannot be specified within bracketed complex pairs.

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

279-2 Too many subscripts specified. Only n are required.

Default: A return will be made to the system.

280-2 Too few subscripts specified. n are required.

Default: A return will be made to the system.

281-2 Zero or negative subscript specified - must be greater than 0.

282-3 Incomplete or illegal NAMELIST variable name (format is name=value).

Default: The remainder of the read will be skipped, and execution of the program will continue with the next statement.

283-2 Subscript larger than maximum set in declaration statement.

Default: A return will be made to the system.

284-2 Subscript less than minimum set in declaration statement.

Default: Control will be returned to the system.

List-Directed I/O Errors

321-1 Variable (or array element) in output list is undefined.

This is a warning.

Default: A field of U's will be written.

322-3 Illegal parameter encountered in call to SIOC.

Default: The rest of the I/O operation will be ignored.

323-2 Complex pair should be terminated by a ')'.

This is a warning.

Default: The error will be ignored and execution resumed.

324-1 Output field width too small.

This is a warning.

Default: A field of *'s will be written.

325-2 Invalid {INTEGER|REAL|COMPLEX} number, illegal character or bad syntax.

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

326-2 Magnitude of REAL number exceeds machine limits: (.53976053469340278 9D-78, .72370055773322621 1D+76)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

327-2 INTEGER*4 number exceeds machine limits: (-2147483648, +2147483647)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

328-2 INTEGER*2 number exceeds machine limits: (-32768, +32767)

If the mode is conversational, the line from the field in error may be reentered from the terminal.

Default: A return will be made to the system.

329-2 Literal string is longer than the element size.

This is a warning.

Default: The literal will be truncated and execution resumed.

330-1 Output record too short for a single variable of this type.

Default: A field of *'s will be written.

Miscellaneous Errors

68-3 FORTRAN-H extended processing is not yet implemented.

Default: System error causes return to system.

69-4 Maximum number of nesting levels in implied DO loop processing exceeded (FORTRAN-H Extended).

70-4 Implied DO loop increment is not positive (FORTRAN-H Extended).

71-4 Implied DO loop ending value is smaller than initial value (FORTRAN-H Extended).

385-3 SETSTA (Setstare) subroutine was called but is no longer available. Now you can use the BACKSPACE statement to backspace any file or device. Unit record equipment may be backspaced 1 record.

Default: The call to SETSTA will be ignored and execution will resume within the program at the next statement.

386-3 SETBLK (Setblock) subroutine was called but is no longer available. It is obsolete -- blocking is provided by the magnetic tape support routines.

Default: The call to SETBLK will be ignored and execution will resume within the program at the next statement.

387-3 FORTRAN I/O statistics collection failure, "FIO:STATFILE" is not available.

This is a system error. (This message will not occur at UM).

388-4 Source code error at ISN n in subroutine s. Consult the listing of your compilation.

Default: Execution will be terminated.

389-3 FORTRAN I/O failure - space management error. Likely cause is no space available. Grab=hhhh.

This is a system error.

390-3 Attempting to empty a DEFINE FILE unit.

Default: The call to EMPTYF will be ignored.

391-2 Attempt to empty a device or a non-existent file.

This occurs in FORTRAN call to EMPTYF. This is a warning.

Default: A return to the program will be made with RC=4 (RETURN 1).

392-2 Hardware error during EMPTY.

This occurs in FORTRAN call to EMPTYF. This is a warning.

Default: A return to the program will be made with RC=8 (RETURN 2).

393-2 EMPTY access not allowed for file.

This occurs in FORTRAN call to EMPTYF. This is a warning.

Default: A return to the program will be made with RC=12 (RETURN 3).

394-2 Locking for modification (EMPTY), will result in a deadlock.

This occurs in FORTRAN call to EMPTYF. This is a warning.

Default: A return to the program will be made with RC=16 (RETURN 4).

395-2 Waiting for file during EMPTY was interrupted - file not emptied.

This occurs in FORTRAN call to EMPTYF. This is a warning.

Default: A return to the program will be made with RC=20 (RETURN 5).

397-3 Attempt to start or unload an improperly nested or nonexistent load module.

Default: Because this error is fatal, control will be returned to the system.

398-3 Illegal call to subroutine "SLITE".

The argument must be from 0 through 4 for SLITE.

Default: Because this error is fatal, control will be returned to the system.

399-3 FORTRAN I/O Failure detected by DEBUG package routine, error return from SIOC.

This is a system error.

400-2 UNLDF may be called only by index: CALL UNLDF(0,INDEX,0) where INDEX is the value returned by LOADF.

This is a warning.

Default: The call to UNLDF will be ignored.

October 1983

THE ELEMENTARY FUNCTION LIBRARY

The elementary function library (EFL) contains the mathematical and implicitly called subroutines usually associated with the FORTRAN IV language. In the FORTRAN language the mathematical routines are called because of an explicit reference to the name of the function in an arithmetic expression. Mathematical routines for the computation of the square root, exponential, logarithmic, trigonometric, hyperbolic, gamma and error functions are provided. The implicitly called routines are invoked to perform complex multiplication and division, and to perform the various exponentiation operations occasioned by the FORTRAN ** operator. Finally, this library also includes the ANSI FORTRAN intrinsic minimum and maximum value functions, and the DREAL and DIMAG functions, which are inexplicably not a part of the IBM FORTRAN library.

The programs contained in this elementary function library are system resident, and are defined in the low-core symbol dictionary named <EFL>. Special loader control cards at the end of the *LIBRARY file cause the symbol <EFL> to be defined; and, if there are still undefined symbols, then this symbol dictionary will be searched.

List of Entry Points by General Function

Absolute Value	CABS, CDABS
Square Root	SQRT, DSQRT, CSQRT, CDSQRT
Common and Natural Logarithm	ALOG, ALOG10, DLOG, DLOG10, CLOG, CDLOG
Exponential	EXP, DEXP, CEXP, CDEXP
Trigonometric Functions	COS, SIN, TAN, COTAN, DCOS, DSIN, DTAN, DCOTAN, CCOS, CSIN, CDCOS, CDSIN
Inverse Trigonometric Functions	ARCOS, ARSIN, ATAN, ATAN2, DARCOS, DARSIN, DATAN, DATAN2
Hyperbolic Functions	COSH, SINH, TANH, DCOSH, DSINH, DTANH
Gamma and Log-gamma Functions	GAMMA, ALGAMA, DGAMMA, DLGAMA
Error Function	ERFC, ERF, DERFC, DERF
Exponentiation	FIXPI#, FRXPI#, FDXPI#, FCXPI#,FCDXI#, FRXPR#, FDXPD#
Complex Operations	CMPY#, CDVD#, CDMPY#, CDDVD#, DREAL[1], DIMAG[1]
Minimum/Maximum Value	MIN0, AMIN0, MIN1, AMIN1, DMIN1 MAX0, AMAX0, MAX1, AMAX1, DMAX1

[1]Since the DREAL and DIMAG functions are not built into the current FORTRAN compilers, they must be explicitly declared as REAL*8 functions.

Mathematical Functions

REAL*4	REAL*8	COMPLEX*8	COMPLEX*16
		CABS[1]	CDABS[1]
SQRT	DSQRT	CSQRT	CDSQRT
EXP	DEXP	CEXP	CDEXP
ALOG	DLOG	CLOG	CDLOG
ALOG10	DLOG10		
COS	DCOS	CCOS	CDCOS
SIN	DSIN	CSIN	CDSIN
TAN	DTAN		
COTAN	DCOTAN		
ARCOS	DARCOS		
ARSIN	DARSIN		
ATAN[1]	DATAN[1]		
ATAN2[2]	DATAN2[2]		
COSH	DCOSH		
SINH	DSINH		
TANH[1]	DTANH[1]		
ERFC[1]	DERFC[1]		
ERF[1]	DERF[1]		
ALGAMA	DLGAMA		
GAMMA	DGAMMA		

FORTRAN Implicitly Called Functions

Complex operations: name(multiplicand-dividend,multiplier-divisor)

COMPLEX*8	COMPLEX*16
CMPY#	CDMPY#
CDVD#	CDDVD#

Exponentiation: name(base,exponent)

Name	Base	Exponent
FIXPI#	INTEGER*4	INTEGER*4
FRXPI#	REAL*4	INTEGER*4
FDXPI#	REAL*8	INTEGER*4
FCXPI#	COMPLEX*8	INTEGER*4
FCDXI#	COMPLEX*16	INTEGER*4
FRXPR#	REAL*4	REAL*4
FDXPD#	REAL*8	REAL*8

ANSI FORTRAN Minimum/Maximum Value

Name	Arguments Mode	Result Mode
MIN0/MAX0	INTEGER*4	INTEGER*4
MIN1/MAX1	REAL*4	INTEGER*4
AMIN0/AMAX0	INTEGER*4	REAL*4
AMIN1/AMAX1	REAL*4	REAL*4
DMIN1/DMAX1	REAL*8	REAL*8

[1]These routines do not recognize any error conditions and never transfer to the error monitor.
[2]These routines require two arguments.

Calling Conventions

The programs contained in the EFL conform to the OS(I) S-type calling convention with variable length parameter list as described in the section "Calling Conventions" in MTS Volume 3, System Subroutine Descriptions, i.e., they expect the FORTRAN linkage convention. This convention requires that the high-order bit of the last parameter address constant be nonzero. The EFL error monitor uses this last argument flag to determine how error situations should be processed; consequently, failure to properly set this flag may result in unexpected results if an error condition is detected. Further, unless specifically mentioned, all elements of the EFL require an 18-fullword (72-byte) save area.

Since all members of the EFL are function-type subroutines, they cannot be meaningfully employed in the FORTRAN CALL statement because the FORTRAN program will ignore the function value returned by these programs. These function subprograms are called whenever the appropriate entry name appears in a FORTRAN arithmetic expression. The following FORTRAN arithmetic assignment statement refers to the mathematical functions COS and SQRT and the implicitly called exponentiation routine FRXPI#:

```
SINX = SQRT(1.-COS(X)**2)
```

Assembly language users may employ the CALL macro, but should specify the optional VL parameter in order to set the last argument flag byte, e.g.,

```
CALL DCOSH,(X),VL
```

The elementary functions return their values as follows:

```
GR0          - INTEGER function
FR0          - REAL function
FR0,FR2      - COMPLEX function
```

Except as noted, the mathematical functions require a single argument of the same mode as the function. The routines in the EFL are subject to specification exceptions when fetching their argument(s) should the boundary alignment be incorrect. The modes INTEGER*4, REAL*4, and COMPLEX*8 require fullword alignment, while REAL*8 and COMPLEX*16 require doubleword alignment. The term INTEGER*4 corresponds to a System/360 fullword integer in the usual twos-complement notation. The term REAL*4 (REAL*8) corresponds to a System/360 short (long) operand floating-point number. The term COMPLEX*8 (COMPLEX*16) refers to two short (long) operand floating-point numbers occupying consecutive storage locations, the number in the higher storage location being the imaginary part of the complex number. The address constant passed to the EFL routine should correspond to the lower storage address, i.e., the REAL part of the complex number.

Error Processing

Error conditions detected by EFL routines are processed in the module ERRMON#. Depending on the optional arguments passed to the elementary function, the error monitor will either resume execution or provide an appropriate error comment and call the subroutine ERROR#.

The vast majority of the EFL programs check the argument to ensure that a valid function value can be computed. For example, the inverse sine and cosine functions are only defined on the interval [-1,1] so that some procedure must be available for handling arguments outside this interval. There are currently three ways in which error conditions detected by an EFL program can be processed:

(1) by using one or more of the optional arguments described below,
(2) by calling the user error monitor, or
(3) by printing an error message on SERCOM and then calling the subroutine ERROR#.

Whenever an elementary function detects an error situation, it generates a default function value and passes control to the EFL error monitor. Although this error monitor is in fact a separate program, it is logically a part of each elementary function and is transparent with respect to the normal linkage conventions.

The EFL error monitor initially attempts to process the optional arguments. If no such arguments were given, or if their processing does not result in the resumption of execution, then the error monitor will formulate an appropriate message. This message is passed, as the sole argument, to the user error monitor or is printed on SERCOM.

With all optional arguments attached, the calling sequence becomes

```
...name(argument(s),count,max-count,f-value)...
```

Since the elementary function names are built into the FORTRAN compiler, it will diagnose as errors any occurrence of these names in which the

number and modes of the arguments do not correspond to its table of definitions. The optional arguments discussed here may be appended to the usual argument list, without objection from the FORTRAN compiler, if the elementary function name is declared in an EXTERNAL statement and its proper mode is explicitly declared. The optional arguments are defined as follows:

count - a fullword integer which is simply incremented by 1. If count is the only optional argument supplied, then execution is resumed with the default function value and return code 4.

max-count - a fullword integer upper bound for the first optional argument, count. If the updated value of count is greater than max-count, then the processing of the optional arguments is suspended. If max-count is the last optional argument supplied and the updated value of count is less than or equal to max-count, execution is resumed with the default function value and return code 4. Otherwise, the final optional argument is processed.

f-value - the mode of this argument must correspond to the mode of the function. Execution is resumed with a function value of f-value and return code 4. Note that this optional argument is processed only if the updated value of count is less than or equal to max-count.

In the above descriptions, the phrase "resume execution" means that it will appear that the elementary function has returned with the indicated function value and return code.

If one of the optional arguments cannot be appropriately accessed, if count > max-count, or if no optional arguments are supplied, then the error monitor will formulate an error message. For the mathematical functions, this error message will take the form

```
name(x.x)  IS UNDEFINED AND HAS BEEN ASSIGNED THE VALUE y.y.
THE DOMAIN OF DEFINITION OF THIS FUNCTION IS dod-message.
```

where "x.x" and "y.y" are decimal representations of the argument and function value, respectively. The "dod-message" is dependent on the elementary function involved, but generally expresses the set of argument values for which the function is defined in the form

```
(x: a < x < k )
```

For example, the GAMMA function "dod-message" is "IS (X: .1381786E-75 < X < 57.57441)".

Messages generated for exponentiation errors take the form:

```
EXPONENTIATION ERROR:  b.b ** e.e IS UNDEFINED AND HAS BEEN
ASSIGNED THE VALUE y.y.  MODE OF THE BASE IS mb, MODE OF THE
EXPONENT IS me.
```

where "b.b", "e.e", and "y.y" are decimal representations of the base, exponent, and result, respectively. The modes "md" and "me" will be one of the following: INTEGER*4, REAL*4, REAL*8, COMPLEX*8, or COMPLEX*16. Generally, exponentiation routines only recognize an error when the base is 0.0 and the exponent is nonpositive; however, the current routines also complain when a real result cannot be properly represented, e.g., 10.**80. In either case, the error monitor dynamically allocates virtual memory space sufficient to generate and assemble this message. The message is generated in the form of a halfword integer length immediately followed by the text of the message.

An elementary function library user error monitor is established by using the CUINFO subroutine. The name and index of the corresponding CUINFO item is 'EFLUEM ' and 183, respectively, while the data is the address of the user error monitor. Thus, to establish a subroutine named UEM as the user error monitor, one could include the following FORTRAN statements in his program:

```
EXTERNAL $UEM$
CALL CUINFO(183,$UEM$)
```

A user error monitor may be eliminated by calling CUINFO with a second argument of zero. The single argument to the user error monitor should be declared as an INTEGER*2 vector, e.g.,

```
SUBROUTINE $UEM$(MSG)
INTEGER*2 MSG(2)
CALL SERCOM(MSG(2),MSG(1),0)
RETURN
END
```

This rather simple example prints the message on logical I/O unit SERCOM, and then resumes execution with the default function value. Since the messages are generally longer than a terminal output line, some of the message will be lost. Unless the user error monitor returns to the EFL error monitor, the virtual memory space allocated by this latter program will not be released.

Finally, if the optional argument processing did not result in the resumption of execution and no user error monitor is established, then the EFL error monitor will provide, on SERCOM, an error message and a trace of the programs in the current linkage chain, i.e., the sequence of programs which have been called, but which have not yet returned to their calling programs. For example, if a main program named MAIN calls a subroutine named SUB, which attempts to compute DLOG(-5.D0), then the linkage chain is SUB, MAIN, and MTS. After providing this information, the error monitor will call the resident system subroutine ERROR#. If a subsequent $RESTART command is issued, execution will resume with the default function value.

Example 1:

```
C PROGRAM TO COMPUTE THE SQUARE ROOTS OF THE
C ABSOLUTE VALUES OF THE NUMBERS READ FROM THE
C INPUT STREAM AND KEEP A COUNT OF THE TOTAL
C NUMBER OF NEGATIVE NUMBERS READ.
      EXTERNAL SQRT
      INTEGER I/0/
10    READ 100,X
      Y = SQRT(X,I)
      PRINT 200,X,Y,I
      GO TO 10
100   FORMAT (E20.8)
200   FORMAT (2E17.9,I5)
      END
```

Example 2:

If the fourth statement in example 1 is replaced by

```
Y = SQRT(X,I,10)
```

then execution will be suspended when the 11th negative argument is passed to SQRT.

Example 3:

```
C PROGRAM TO TEST THE IDENTITY
C COS(X)**2 + SIN(X)**2 = 1
C FOR VALUES OF X READ FROM THE INPUT STREAM.  THE
C DSIN AND DCOS ROUTINES ARE UNDEFINED FOR X > PI*2**50,
C BUT THE DEFAULT VALUES CHOSEN GUARANTEE THE IDENTITY.
      EXTERNAL DCOS,DSIN
      REAL*8 DCOS,DSIN,X,ONE
10    IER = 0
      READ 100,X
      ONE = DCOS(X,IER,IER,0.D0)**2+DSIN(X,IER,IER,1.D0)**2
      PRINT 100,IER,ONE
      GO TO 10
100   FORMAT (E20.8)
200   FORMAT (I3,E17.9)
      END
```

Example 4:

The use of the following parameter list would guarantee that the elementary function would always denote error situations by a return code of 4.

```
          DC    A(argument),XL1'FF',AL3(ERRCNT)
ERRCNT    DC    F'0'
```

In addition, the word ERRCNT would be automatically updated to maintain a count of the total number of errors.

Mathematical Functions

The following descriptions of the mathematical functions are limited to error conditions which may arise in these programs. These routines are consistent with the FORTRAN IV library functions currently distributed with the System/360 Operating System and have been documented by IBM in their publication, IBM System/360 Operating System FORTRAN IV Library - Mathematical and Service Subprograms, form GC28-6818.

Square Root

Because SQRT and DSQRT are specifically defined as real-valued functions, they are not defined for negative real arguments. The default function value computed when the argument is negative is the square root of the absolute value of the argument.

Common and Natural Logarithm

The real-valued logarithm functions ALOG, ALOG10, DLOG, and DLOG10 are not defined for negative arguments since the logarithm of a negative number is complex, i.e., if $x<0$ then $\ln(x) = \ln(|x|) - i \cdot Pi$. The default function value is the logarithm of the absolute value of the argument.

All of the logarithmic functions are undefined for an argument of zero, which is a pole of the logarithm function. Appropriately, the default function value is negative machine infinity, i.e., roughly $-.7237005 \cdot 10^{76}$.

Exponential

The real-valued functions EXP and DEXP can be properly defined only in the interval [-180.2182,174.67308] because of the range restrictions imposed by the floating-point representation. The largest positive number representable in System/360 floating-point form is $16^{63} \cdot (1-16^{-14})$, and the natural logarithm of this number is approximately 174.67308. Similarly, -180.2182 is the logarithm of the smallest positive number, 16^{-65}. The actual domains are as follows:

EXP (hex)	-B4.37DF	AE.AC4F
DEXP (hex)	-B4.37DEFFFFFFFF	AE.AC4EFFFFFFFF
EXP (dec)	-180.218246	174.673080
DEXP (dec)	-180.218246459960934	174.673080444335934

If the argument exceeds the upper limit, the default function value is machine infinity. If the argument is less than the lower limit, the default function value is zero; however, this situation is

regarded as an error if and only if underflow exceptions are enabled by the program mask.

It should be noted that the domain of the exponential functions is slightly smaller than the range of the corresponding natural logarithm functions. Hence, the expressions EXP(ALOG(X)) and DEXP(DLOG(X)) are not computable for values of X extremely close to the ends of the machine range.

The complex-valued functions CEXP and CDEXP have an analogous domain restriction on the real part of the complex argument and an additional restriction on the imaginary part. Whether the complex argument satisfies the domain restrictions or not, the value of the CEXP(x+i•y) will be

EXP(x)•[COS(y)+i•SIN(y)]

and that of CDEXP(x+i•y) will be

DEXP(x)•[DCOS(y)+i•DSIN(y)]

Trigonometric Functions

The domain restrictions of the real-valued trigonometric functions COS, SIN, TAN, COTAN, DCOS, DSIN, DTAN, and DCOTAN are imposed to maintain accuracy. These functions are computed by reducing the argument to the interval [-Pi/4,Pi/4] by using the periodicity of these functions. For very large arguments this reduction yields so few significant digits in the reduced argument that meaningful computation of the function value is impossible. The single-precision functions require

$|x| < 2^{18} \cdot Pi$ = C90FD.9 = 823549.563

while the limit for the double-precision functions is

$|x| < 2^{50} \cdot Pi$ = C90FD9FFFFFFF.F = 3537118706008063.94

The default function value is uniformly zero.

In addition, the tangent and cotangent functions will object if the argument is too close to one of their singularities to maintain accuracy or if the function value would exceed the machine range. In these situations, the default function value is machine infinity with the sign of the argument.

The complex sine and cosine functions CCOS, CDCOS, CSIN, and CDSIN can be defined as

sin(x+i•y) = sin(x)•cosh(y)+i•cos(x)•sinh(y)

cos(x+i•y) = cos(x)•cosh(y)+i•sin(x)•sinh(y)

These formulas illustrate why a trigonometric-type domain restriction is applied to x, and an exponential-type domain restriction to y. The default function value is derived from the default values supplied by the appropriate sine, cosine, and exponential routines, where cosh(y) and sinh(y) become machine infinity divided by 2 when |y| is too large.

Inverse Trigonometric Functions

The domain of the inverse sine and cosine functions ARCOS, ARSIN, DARCOS, and DARSIN is the range of the sine and cosine functions, i.e., [-1,1]. Outside this interval, the default function value is zero.

The inverse tangent routines ATAN2 and DATAN2 are undefined only for the argument pair (0.,0.), for which the default function value is zero. In effect, given the argument pair (y,x), these routines compute the principal value of the argument of the complex number x+i•y.

Hyperbolic Functions

The value of the hyperbolic sine and cosine of x exceed the range of the machine when |x| approaches the logarithm of machine infinity. Specifically, the domain of the COSH and SINH routines is described by

$$|x| \leq \text{AF.5DC0} = 175.366211$$

and that of DCOSH and DSINH by

$$|x| \leq \text{AF.5DC0FFFFFFFF} = 175.366226196289059$$

The default function value is machine infinity with the appropriate sign.

Gamma and Log-Gamma Functions

Like the exponential function, these functions exceed machine range outside their domains of definition and have a default function value of machine infinity. The specific hexadecimal intervals of definition are

GAMMA	$[.100001 \cdot 16^{-62}, 39.930D]$
DGAMMA	$[.100001 \cdot 16^{-62}, 39.930CFFFFFFFF]$
ALGAMA	$[0, .184D30 \cdot 16^{62}]$
DLGAMA	$[0, .184D2FFFFFFFFF \cdot 16^{62}]$

while in decimal these intervals become

GAMMA	$[.138178829 \cdot 10^{-75}, 57.5744171]$
DGAMMA	$[.13817882865895404 \cdot 10^{-75}, 57.5744171142578089]$

ALGAMA [0,.429370581•10⁷⁴]
DLGAMA [0,.4293705810082411 43•10⁷⁴]

Implicitly Called Functions

Complex Arithmetic Operations

CMPY# (COMPLEX*8-multiplicand,COMPLEX*8-multiplier)
CDVD# (COMPLEX*8-dividend,COMPLEX*8-divisor)
CDMPY# (COMPLEX*16-multiplicand,COMPLEX*16-multiplier)
CDDVD# (COMPLEX*16-dividend,COMPLEX*16-divisor)

Algorithm:

The multiplication algorithm takes the form

$$(x+iy)\cdot(u+iv) = (x\cdot u-y\cdot v)+i(v\cdot x+u\cdot y)$$

The division algorithm is likewise direct and takes the form

$$\frac{(x\cdot u+y\cdot v)+i(u\cdot y-v\cdot x)}{u\cdot u+v\cdot v}$$

with appropriate scaling of the divisor u+iv to avoid floating-point overflow or underflow of the denominator.

Error Conditions:

Both underflow and overflow exceptions may occur during the formation of the final result. Zero-divide exceptions may also occur, but only if u=v=0.

Exponentiation

FIXPI# (INTEGER*4-base,INTEGER*4-exponent)
FRXPI# (REAL*4-base,INTEGER*4-exponent)
FDXPI# (REAL*8-base,INTEGER*4-exponent)
FCXPI# (COMPLEX*8-base,INTEGER*4-exponent)
FCDXI# (COMPLEX*16-base,INTEGER*4-exponent)

Algorithm:

Though each of these routines differ in some way, they all obtain the result by the successive squaring algorithm. This algorithm exploits the binary representation of the integer exponent to compute R=B**I in the following steps:

(1) Initialize R=1., S=B and k=0.
(2) If the k-th bit of |I| is 1, replace the current value of R by R•S.
(3) If one or more of the unexamined bits of |I| is 1,

replace S by S•S, increment k by 1, and return to step (2); otherwise, R=B**|I|.

The FIXPI# routine recognizes a number of special cases, none of which actually require any computation.

Base:	≠0	1	-1	-1	≠0
Exponent:	0	any	even	odd	<0
Result:	1	1	1	-1	0

During the course of the algorithm, the result is not range-checked. Consequently, the result is valid only if it is in machine range, i.e., less than 2^{31} = 2,147,483,648.

The FRXPI# and FDXPI# routines form B**|I|, and then divide this result into 1.0 if I is negative. Both routines recognize a nonzero base and zero exponent as a special case having value 1. These routines range-check the result as it is being formed, and will invoke error processing if B**|I| or B**I are not machine representable. In FRXPI#, B**|I| is formed in double precision.

In the FCXPI# and FCDXI# routines, a negative exponent causes the base to be inverted before the successive squaring algorithm is applied. Both routines recognize a nonzero base and zero exponent as a special case having value 1. These routines do not range-check the result and are subject to underflow and overflow exceptions. Note that if underflow exceptions are masked off (by default, they are not masked off), the complex base is extremely small, and the exponent negative, a zero-divide exception may occur when the base is initially inverted. These routines use the end of the save area for scratch storage.

Error Conditions:

All of these routines recognize a zero base and nonpositive exponent as an error. In addition, the FRXPI# and FDXPI# routines will invoke error processing if either B**|I| or the final result is outside machine range. In all cases, the default function value is zero.

FRXPR# (REAL*4-base,REAL*4-exponent)
FDXPD# (REAL*8-base,REAL*8-exponent)

Algorithm:

The result is obtained by using the appropriate logarithm and exponential routines, i.e.,

e **(exponent•ln(base))

These routines recognize as a special case the combination of a zero base and positive exponent. If exponent•ln(base) < 0, the final result is not in machine range, and underflows are masked off, these routines may return a result of zero.

Error Conditions:

The combination of a zero base and nonpositive exponent causes error processing to be invoked with a default value of 0. Denote the base by B and the exponent by E. If B<0 , but |B|**E is in machine range, the default function value is |B|**E. If E•ln(|B|) is within machine range, but the result is not, the default function value will be zero if E•ln(|B|) <0, and machine infinity if E•ln(|B|)>0. If E•ln(|B|) is not in machine range, the default function value is zero.

DREAL and DIMAG Functions

```
DREAL          (COMPLEX*16-variable)
DIMAG          (COMPLEX*16-variable)
```

Algorithm:

Although these routines are described in the IBM FORTRAN language manual, the currently available FORTRAN compilers do not recognize these names as anything special. Consequently, it is necessary to explicitly declare them as REAL*8 functions. Otherwise, they will be assigned the default mode of REAL*4.

These routines are extremely trivial, consisting of the bare minimum of three instructions. Only general register 1 and floating-point register 0 are altered by these routines, and a save area is not required.

Error Conditions:

These routines are subject to specification exceptions since they assume the argument is doubleword-aligned.

ANSI Minimum/Maximum Value Functions

```
MIN0/MAX0      (INTEGER*4-variable,...)
AMIN0/AMAX0    (INTEGER*4-variable,...)
MIN1/MAX1      (REAL*4-variable,...)
AMIN1/AMAX1    (REAL*4-variable,...)
DMIN1/DMAX1    (REAL*8-variable,...)
```

Algorithm:

These routines are identical in structure, accepting a variable number of arbitrary arguments of the appropriate mode and recognizing no error situations. The resultant modes of these

entry points are determined by the first character of the function names as follows: M=INTEGER*4, A=REAL*4, and D=REAL*8. The number of arguments processed is determined by the last argument flag. This flag is set automatically by FORTRAN; however, assembler users must make sure this flag is properly set; otherwise, an addressing or protection exception may occur.

October 1983

FREAD/FWRITE: FREE-FORMAT I/O SUBROUTINES

INTRODUCTION

Free-format input/output simplifies data transmission in the sense that the user is freed from FORMAT statement restrictions.

For free-format input, rather than having to enter data within certain column boundaries, the user need only ensure that adjacent data fields are separated by a delimiter. Normally, this means adjacent data fields must be separated by one or more blanks and/or a comma.

Free-format output simplifies printing of output that does not require complex formatting, such as interactive query messages. The user is freed from FORMAT statement restrictions, and can see at a glance what is being printed.

USING FREAD TO INPUT DATA

FREAD is a free-format input routine designed to read

- integer numbers
- real numbers
- logical values
- MTS line numbers
- character strings
- hexadecimal strings
- octal strings
- binary strings

from any I/O unit or user-supplied buffer. FREAD offers the advantages of being faster and more responsive than the standard FORTRAN READ statement; for example, an incorrect data value needs only to be reentered.

FREAD is restricted for use with files and devices with input record lengths ≤ 255 bytes.

Two types of calling sequences are defined for FREAD: general (or data) calling sequences and special (or control) calling sequences.

General Calling Sequence

The general FREAD calling sequence is:

```
CALL FREAD(unit,format,list,&n1,&n2,&n3)
```

where:

unit is one of,

'GUSER'	to read from the MTS I/O unit GUSER.
'SCARDS'	to read from the MTS I/O unit SCARDS.
0-99	to read from one of the MTS I/O units 0 through 99. Note that no default unit assignments are made.
'PAR'	to read from the PAR field of the $RUN command.
'*'	to continue reading from the same line or buffer as on the previous call to FREAD.
FDUB-ptr	as returned by an MTS subroutine such as GETFD (if the NOFDUB option is specified).

format is a string of character information or an array containing such information (much like a FORMAT string) indicating how many and what types of variables are to be read. As in the FORMAT string, individual type-codes within the type string must be separated by commas, and several such type codes may be enclosed in parentheses to form a type-group. There is no limit to the level to which groups may be nested. Repeat counts may prefix both groups and individual type-codes. Type strings must be terminated either by a colon or an ellipsis '...'. A type string terminated by an ellipsis causes the immediately preceding group or type-code to be treated as if it had an infinite repeat count.

The following type-codes are recognized:

INTEGER,INTEGER*4,I,I*4	for INTEGER*4 numbers
INTEGER*2,I*2	for INTEGER*2 numbers
REAL,REAL*4,R,R*4	for REAL*4 numbers
REAL*8,R*8	for REAL*8 numbers
LOGICAL,LOGICAL*4,L,L*4	for LOGICAL*4 numbers
LOGICAL*1,L*1	for LOGICAL*1 numbers
LINENUMBER,#	for MTS line numbers
STRING,S,M	for character strings
HEX,H,Z	for hexadecimal strings
OCTAL,O	for octal strings
BINARY,B	for binary strings

If the type code is not present (i.e., if the type string is set to a colon), FREAD will read a line and return to the calling program without transferring any data.

list is the list of variables or arrays into which the numbers and/or strings are to be read.

&n1 (optional) is the FORTRAN statement number to transfer to if FREAD executes a RETURN 1 to the calling program.

&n2 (optional) is the FORTRAN statement number to transfer to if FREAD executes a RETURN 2 to the calling program.

&n3 (optional) is the FORTRAN statement number to transfer to if FREAD executes a RETURN 3 to the calling program.

See the section "Error Recovery" below for further details on statements labels to transfer to if FREAD executes a RETURN 1, 2, or 3 statement to the calling program.

The following examples illustrate calls to the FREAD subroutine.

```
CALL FREAD(0,'I,I,R:',I,J,X)
```

reads two integers and a real number from logical I/O unit 0.

```
CALL FREAD('SCARDS','R:',X)
```

reads a single real number from logical I/O unit SCARDS.

```
CMDLEN=20
CALL FREAD('*','S:',CMD,CMDLEN)
```

reads a character string of up to 20 characters from the next field in the current record.

Reading Numeric Data

This section defines how to read INTEGER and REAL data. FREAD can read into one or more single variables or one or more singly dimensioned arrays. Special actions are required to use FREAD to input data into multiply dimensioned arrays. Normally, there is a one-to-one correspondence between the type-codes specified in the "format" and the parameters specified in the "list".

The type-codes for INTEGER data depends on the length of the integer variable: use I, I*4, INTEGER, or INTEGER*4 for INTEGER*4 variables; or I*2, INTEGER*2 for INTEGER*2 variables. For example, the following (equivalent) calls will read two values from unit 5 into the variables I and J:

```
CALL FREAD(5,'INTEGER,INTEGER*2:',I,J)
CALL FREAD(5,'I,I*2:',I,J)
```

```
CALL FREAD(5,'I:',I)
CALL FREAD('*','I*2:',J)
```

The type-code to indicate a REAL (i.e., floating-point) conversion also depends on the length of the variable: use R, R*4, REAL, or REAL*4 for REAL*4 variables; use R*8 or REAL*8 for double-precision variables. For example, the following (identical) calls will read two floating-point numbers into variables X and Y:

```
CALL FREAD(5,'REAL,REAL*8:',X,Y)
CALL FREAD(5,'R,R*8:',X,Y)
```

It is not possible for FREAD to read an entire multiple-dimensioned array in a single call. It is, however, possible to read an entire vector (i.e., a one-dimensioned array) with one call by using the VECTOR type-code and specifying the number of elements to be read.

To indicate that a vector is to be read, the word 'VECTOR' (or 'V') is appended to the type string:

```
CALL FREAD(0,'INTEGER VECTOR:',I,10)
```

As well, an INTEGER variable which gives the number of elements to be read must be supplied in the parameter list. Thus, for each vector to be read there must be two parameters supplied in the call to FREAD.

The following examples illustrate the differences between calls to FREAD and standard FORTRAN:

```
via FFIO:       CALL FREAD(5,'2I:',I,J)
                CALL FREAD('*','R,R*8:',X(I,J),Y(I,J))
via FORTRAN:    READ(5,1) I,J,X(I,J),Y(I,J)
              1 FORMAT(2I3,F12.3,D16.7)

via FFIO:       CALL FREAD(5,'2(I,R):',I1,R1,I2,R2)
via FORTRAN:    READ(5,1) I1,R1,I2,R2
              1 FORMAT(2(I5,F10.2))

via FFIO:       CALL FREAD(5,'INTEGER VECTOR:',I,N)
via FORTRAN:    READ(5,1) (I(J),J=1,N)
              1 FORMAT(10I3)

via FFIO:       CALL FREAD(5,'REAL*8 VECTOR:',X,N)
via FORTRAN:    READ(5,1) (X(J),J=1,N)
              1 FORMAT(10F9.3)
```

Reading Character Data

Character input corresponds to FORTRAN's A-format.

There are two format codes for CHARACTER input: S (or STRING) and M. The S type-code causes the next field to be moved unchanged into the variable specified. By default, the next field begins with the first nonblank character and is terminated by a blank. This can be changed by using the DELIMITER option.

The M type-code causes the entire next record to be moved unchanged into the variable specified. Since M specifies that the entire record is to be read, the length is ALWAYS returned, regardless of the setting of the LENGTH option. Thus, a variable must always be used for the length argument and must be first set to the maximum length of the variable. Note that this implies the argument must be a variable.

Since strings have variable lengths, it is necessary to specify the length of the string expected. Thus, variable list parameters for S or M type-codes must appear in pairs: the first being the location to which the string is to be moved, and the second being the length of the variable (in bytes).

The actual length of a string read using "S" can be obtained by setting the LENGTH option to ON. In this case, the length argument must be a variable (not a literal).

Strings are placed left-justified in the variable. If it is longer than the length, the string is truncated on the right, whereas if it is shorter, it is padded on the right with blanks.

The VECTOR type is not available with S or M type-codes.

The following examples illustrate the reading of character input.

```
CALL FREAD(5,'STRING:',WORD,4)

LOGICAL*1 TITLE(132)
INTEGER TITLEN/132/
CALL FREAD('SCARDS','M:',TITLE,TITLEN)
```

Reading Other Types Of Data

Integer, real, and character data are the most common data types, but FREAD allows several other types as well. These include LOGICAL, HEXADECIMAL, OCTAL, BINARY, and MTS line numbers. Complete descriptions of these data types are given in the section "Data Descriptions" below.

The type-code for LOGICAL input is L, L*4, LOGICAL, or LOGICAL*4 for logical variables of length 4; for LOGICAL*1 variables use L*1 or LOGICAL*1. Any string is an acceptable logical value. If T or TRUE is found, then TRUE is returned; otherwise, the value is FALSE. There must be one parameter in the variable list for each logical type-code specified.

The type-code for HEXADECIMAL input is HEX, H, or Z. Hexadecimal input is treated in a similar fashion to STRING input, i.e., each H type-code requires two parameters in the variable list: one identifying the location to put the hex string, and one indicating the length of the string.

The type-code for reading MTS line numbers is LINENUMBER or #. Only one parameter in the variable list is required for each type LINENUMBER in the format.

The type-code for OCTAL input is OCTAL or O. Octal input is treated in a similar fashion to STRING input, i.e., each O type-code requires two parameters in the variable list: one identifying the location to put the octal string, and one indicating the length of the string.

The type-code for BINARY input is BINARY or B. Binary input is treated in a similar fashion to STRING input, i.e., each B type-code requires two parameters in the variable list: one identifying the location to put the binary string, and one indicating the length of the string.

Reading from a User-Supplied Buffer

The user can provide an input buffer rather than have FREAD read a line. Thus, the user is able to do "memory-to-memory" reading using the special entry point FREADB. User-provided input buffers must adhere to the following rigid format: a halfword buffer length, followed by a halfword buffer index, followed by the buffer proper. The buffer length is the length of the buffer proper in bytes (not including the length and index). The buffer length must be declared INTEGER*2, and it must fall within the range $0 \leq \text{length} \leq 255$. The buffer index is the relative location within the buffer proper at which processing is to begin. The buffer index also must be declared INTEGER*2, and it must fall within the range $1 \leq \text{index} \leq \text{length}$. The buffer index is ignored when the buffer length is 0.

The following example shows how two real numbers 3.0 and 4.0 can be read from a user-supplied buffer,

```
INTEGER*2 BUFFER(6)
DATA BUFFER(3)/'3.'/, BUFFER(4)/'0'/
DATA BUFFER(5)/'4.'/, BUFFER(6)/'0'/
BUFFER(1)=7
BUFFER(2)=1
CALL FREADB(BUFFER,'2R:',X,Y)
```

Note that if the buffer index, BUFFER(2), had been assigned the value 5 prior to calling FREAD, then only one number would have been read (4.0).

If the BUFFER option is used, FREAD will return its current input buffer together with its length and index in an identical format to that shown above.

For user-supplied buffers, the buffer index is normally not modified by FREADB. However, when the UPDATE option is set to ON, FREADB will update the buffer index to correspond to the next field in the input buffer before returning to the calling program. When the buffer index exceeds the buffer length, there are no more data fields left in the buffer. It is the responsibility of the calling program to check this condition.

Error Recovery

By default, FREAD returns to the calling program when it has successfully read the required variables, or returns via the RETURN 1 statement if there were not enough values for the variables specified.

This default action of FREAD regarding exits can be modified by the ENDLINE, ENDFILE, and/or ERROR options.

End-of-File Exits

When an end-of-file condition is encountered, other than that encountered in response to an error recovery prompt, FREAD returns to MTS command mode so that, for example, I/O units may be reassigned. A subsequent $RESTART command will cause the reading of another data line from the same unit.

Optionally, according to the setting of the ENDFILE option, the program that called FREAD may regain control in the event of an end-of-file.

End-of-Line Exits

An end-of-line condition occurs when FREAD is requested to read more data fields than there are fields left on a line. When an end-of-line condition is encountered, FREAD will RETURN 1 to the calling program after filling unsatisfied variables with zeros or blanks (unsatisfied character string buffers are filled with blanks, all other types are filled with zeros). The actual number of fields found is returned by FREAD as a real number and as an integer function value. The number of fields found is also available through the NUMBER option. The filling of unsatisfied variables with zeros or blanks can be suppressed by setting the NOFILL option to ON.

Optionally, according to the setting of the ENDLINE option, FREAD will RETURN 0 through 3 to the calling program, or return to MTS command mode, or read additional lines until all variables have been satisfied. The reading of additional lines until all variables have been satisfied is known as "stream input." A strong delimiter appearing as the last nonblank character on a line or an end-of-file is the only method of terminating stream input before all variables have been satisfied.

Error Exits

The type of error recovery that occurs depends upon whether a job is being done in conversational mode or in batch mode. When an incorrect data field is encountered in conversational mode, an error message is printed on SERCOM (usually the terminal) and the user is prompted for replacement data from GUSER (usually the terminal).

Suppose, for example, an integer number was being read and the user inadvertently entered "2.0" instead of "2". Then the following sequence of lines would be printed by FREAD on SERCOM,

```
?INVALID INTEGER NUMBER: "2.0"
?ILLEGAL CHARACTER         $
?ENTER REPLACEMENT NUMBER,
 OR RE-ENTER REST OF LINE FROM POINT OF ERROR,
 OR "MTS"
?
```

Usually it is desirable to enter a "replacement field" when an incorrect number has been entered, whereas usually it is desirable to "reenter rest of line from point of error" when an incorrect delimiter has been entered.

If two or more fields are entered or if one field followed by a nonblank delimiter is entered, then FREAD assumes that "rest of line from point of error" was entered. In all other cases FREAD assumes a "replacement field" was entered.

If "MTS" or an end-of-file is entered, then FREAD returns to MTS command mode. A subsequent $RESTART command will cause another prompt for replacement data.

If a null line is entered or if a delimiter is entered as the only character on an otherwise blank line, then FREAD assumes a null replacement field has been entered. In all other cases FREAD assumes that a null replacement field has not been entered.

Note that if "rest of line from point of error" is entered, it has the immediate effect of actually editing the input buffer. Thus, the input buffer may become shorter or longer. Similarly, if a "replacement field" is entered, then the input buffer is also

edited--but only if the "replacement field" is correct. If the "replacement field" is not correct, FREAD ignores it and prints another prompt for replacement data. In all cases a check is made to see if the edited input buffer would be longer than 255 bytes; if it would be, then the editing is suppressed and the prompt is repeated.

When incorrect data are encountered in batch mode, error messages are also printed on SERCOM, but the prompt for replacement data is not issued; instead, the user is signed off.

The action taken by FREAD when a conversion error occurs can be modified by a call to FREADC that sets the ERROR option. Many different recovery strategies are available.

The verbosity of error messages may be controlled by a special call that sets the VERBOSITY option. From 0 through 3 error message lines per error may be printed; the most verbose forms include the I/O unit and the file or device on which the error occurred.

Use of the INFORMATION option, the REREAD option, the BUFFER option, the LASTDELIMITER option, and the TYPE option allow a program calling FREAD to do its own error processing.

USING FWRITE TO OUTPUT DATA

FWRITE has several advantages over standard FORTRAN I/O. These include the following:

(1) Interspersing text and numeric data without leaving large gaps in the text.
(2) Allowing the user to create a message in pieces, giving greater flexibility in composing the message.
(3) Printing several lines of straight text. FWRITE puts just enough on each print line to be useful without needing to count characters.
(4) Since no format statements are required, it becomes easier to print lines of data.

General Calling Sequence

The general FWRITE calling sequence is:

```
CALL FWRITE(unit,format,list)
```

where:

unit is one of,

'SERCOM' to write to the MTS I/O unit SERCOM.
'SPRINT' to write to the MTS I/O unit SPRINT.
'SPUNCH' to write to the MTS I/O unit SPUNCH.
0-99 to write to one of the MTS I/O units 0 through 99.

format defines the output conversions to be performed. The format consists of characters, which are printed without conversion, and type-codes enclosed by '<' and '>', which identify the type of conversion to be performed. If more than one type-code is specified, a comma is used to separate the two. On output, numbers are separated by one blank. The format is terminated by one of two characters:

';' implies the output line is incomplete; a further call to FWRITE will be made to add to the line.
':' implies the output line is now complete and should be written.

Note: Since the programmer does not have control over the placing of literal strings it is recommended that the terminating ":" or ";" always be followed by a blank. This removes the possibility of having FWRITE find, for example "::" which would cause it to continue searching for a terminator.

list defines the location(s) of the data to be converted for output.

Writing Numeric Data

Numeric data consists of INTEGER and REAL numbers. FORTRAN FORMAT equivalents are Iw, Ew.d, Fw.d, Gw.d, Dw.d.

The format identifier for INTEGER data depends on the length of the integer value. For INTEGER*4 values, use I, INTEGER, I*4, or INTEGER*4; for INTEGER*2 values, use I*2 or INTEGER*2. For example:

```
CALL FWRITE(unit,' Value of J = <I>.: ',J)
```

Assuming J=10, this produces:

```
Value of J = 10.
```

The format identifier for REAL data also depends on the length of the real variable: for REAL*4 values, use R, REAL, R*4, or REAL*4; for REAL*8 values, use R*8 or REAL*8. For example:

```
CALL FWRITE(unit,' Value of X = <REAL>.: ',X)
```

Assuming X=10.99, this produces:

```
Value of X = 10.99.
```

Writing Character Data

Character output is achieved in FORTRAN using A-format.

The format identifier for CHARACTER data S (STRING) or M. A second parameter is required to define the length of the string to be written, if the length is not specified. In actual fact, M only exists to be compatible with FREAD.

Trimming of blanks from both ends of the string is done by making the length negative (the absolute value of length is then used to determine the true length of the string).

The following example could be used for those problems that require the name of the data file to be read in.

```
LOGICAL*1 NAM(17)
INTEGER LEN/17/
CALL FWRITE(6,' Input from file "<STRING>": ',NAM,-LEN)
```

Since the maximum length of a file name is 17 characters, NAM must be set large enough to contain such a name. However, the output line looks better without leading and trailing blanks, so -LEN is specified. The length can also be specified in the format, e.g.,

```
CALL FWRITE(6,'<S*10>: ',NAME)
```

will write the first ten characters of the variable NAME.

Writing Other Types of Data

To write data in HEXADECIMAL use the H, HEXADECIMAL, or Z-format identifier. As for character strings, two parameters are required in the variable list: one to indicate the location of the data, and one to indicate the length of the string. The second parameter can be omitted if the length is specified in the format code. For example, the following calls print the contents of X in hexadecimal:

```
CALL FWRITE('SPRINT',' X=<Z>: ',X,4)
CALL FWRITE(6,' X=<R> is <Z*4> in hexadecimal: ',X,X)
```

To write MTS line numbers, use the LINENUMBER or # format identifier. The following prints the variable LNR as "123.":

```
LNR = 123000
CALL FWRITE(6,' <#>: ',LNR)
```

Special Controls

While FWRITE does not provide all the capability of FORTRAN FORMATs, there are times when simple formatting is useful. Currently, FWRITE supports tabbing and several spacing controls.

Tabbing

To make placement of variables on an output line easier a tab command is provided. The format specifier is T and the column specification can be a constant or a variable. For example, the following two calls are identical:

```
CALL FWRITE(6,'<T10> <I>: ',INTGER)
CALL FWRITE(6,'<T> <I>: ',10,INTGER)
```

If a tab value is given which would position the next output character before the current position, a new output line is started.

Fixed Formats

FWRITE defaults the number of digits to print depending on the variable type. For INTEGER, the field width is as wide as necessary to print the value. For REAL*4, the number of digits to the right of the decimal point is set to 7; for REAL*8, is set to 14.

The field width can be changed by specifying a particular format to use. This is done by following the format identifier with a format enclosed in parentheses: e.g. R*4(F10.2) or I(I5). The formats available are I, F, E, D, or G. FORTRAN format rules are followed except that P (scale) formats are not allowed.

Spacing

There are two methods of forcing spacing on the output page. First, the slash "/" operator, inserted anywhere in the text, causes the current output line to be printed and a new line started. For example,

```
CALL FWRITE(6,'On first line/on second line: ')
```

To actually print the slash "/" character, two consecutive two consecutive "/"s must be specified in the phrase. Second, total control over carriage control is provided by the FWRITC entry point and the CC option. If the CC option has been set to ON, then all output is written with carriage control (the default is that all output is written with CC=DEFAULT which is the current setting of the MTS @CC FDname modifier which defaults on). In this case, the user must ensure that each output line has a valid carriage control character. For example,

```
CALL FWRITC(6,'CC=ON;')
CALL FWRITE(6,'1This line appears at top of page: ')
```

Writing To A User-Supplied Buffer

It is possible by calling the FWRITB entry point, to perform memory-to-memory output. FWRITB is called in exactly the same way as FWRITE, except the unit parameter is replaced by two parameters which specify the memory buffer to "write" the record to, and the length of the buffer. FWRITB behaves the same way as FWRITE except that it returns the output record rather than printing it. For example,

```
LOGICAL*1 CMD(80)
LEN=80
CALL FWRITB(CMD,LEN,'$RES <I>=<S>: ',10,'FILE',4)
```

On return from the FWRITB call, CMD would contain '$RES 10=FILE' and LEN would contain 12. The remaining 68 bytes of CMD would be set to blanks. The subroutine CMDNOE could then be called with the variables CMD and LEN. FWRITB always returns the length of the record "written", and for this reason the length parameter must always be a variable and not a constant. If an attempt is made to generate a message greater than LEN, a RETURN 1 will be made; in this case, the contents of the buffer are unpredictable.

The length parameter may be either an INTEGER*4 or INTEGER*2 variable.

INPUT AND OUTPUT OPTIONS

Options may be specified as input (via FREADC) or as output (via FWRITC). For example, calling FREADC with the LINENUMBER option specifies an indexed read operation for FREAD, and similarly calling FWRITC with the LINENUMBER option specifies an indexed write operation for FWRITE.

The following modifier options may be specified both for input and output:

CC, UC, LC, TRIM, IC, MCC

Note that these options are actually used to set the corresponding MTS I/O FDname modifiers for calls to FREAD and FWRITE. The description of these options corresponds to the descriptions of the similarly named I/O FDname modifiers which are described in Appendix A to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System.

The format for the modifier options is

```
CALL FREADC([unit,]'option={ON|OFF|DEFAULT};')
```

or

```
CALL FWRITC([unit,]'option={ON|OFF|DEFAULT};')
```

For example, UC=ON converts to uppercase while UC=OFF or UC=DEFAULT does not convert to uppercase unless the user applies the @UC FDname modifier on the $RUN command, e.g.,

```
$run program 5=file@UC
```

For example,

```
CALL FREADC(8,'UC=ON;')
```

will convert all input read from unit 8 to uppercase. Input read from other units and all output will not be converted.

Special Input Options for FREADC

Several special input options may be specified for the FREAD subroutine. To set an input option, either of the following two forms may be used:

```
CALL FREADC('option=value;')
```

```
CALL FREADC('option;',value)  or  CALL FREADC('option=?;',value)
```

where "option" specifies the option to be used (with at least the first three characters given) and "value" specifies the value of the option. The first argument must end with ";". "value" may be any of the following:

(1) DEFAULT, e.g., ERROR=DEFAULT
(2) ON (TRUE) or OFF (FALSE), e.g., NULL=OFF
(3) a numerical value, e.g., VERBOSITY=3

(4) a string constant, e.g., JUSTIFY=RIGHT
(5) a string constant enclosed in primes, e.g., PREFIX='-'

The second form should be used when the value is not DEFAULT, a logical value, or numerical value, or a string constant. For example, the MTSLNR option must have the second parameter to store the last line read. According to which option is being used, the second parameter may be

(1) the logical value .TRUE. (fullword 1) or .FALSE. (fullword 0), e.g.,

```
CALL FREADC('LONG=?;',.TRUE.)
```

.TRUE. and .FALSE. correspond to ON and OFF, respectively.

(2) a numerical constant or value to be stored, e.g.,

```
CALL FREADC('MTSLNR;',LNBR)
```

(3) a string value, e.g.,

```
CALL FREADC('ENDLINE;','STREAM')
```

(4) all others (see the descriptions of each option below)

The second argument may not be required in some cases, e.g.,

```
CALL FREADC('SAVE;')
```

If the second argument or value is not specified, DEFAULT is assumed.

The FREADC subroutine may be used to set input options for a specific logical I/O unit by calling it using either of the following two forms:

```
CALL FREADC(unit,'option=value;')

CALL FREADC(unit,'option;',value)
```

where "unit" may be any of the following:

```
0-99      to read from the MTS I/O units 0 through 99.
'GUSER'   to read from the MTS I/O unit GUSER.
'SCARDS'  to read from the MTS I/O unit SCARDS.
'*'       to read from the previous MTS I/O unit.
```

Current options that can be used in this form are

```
LINENUMBER, MTSLNR, BUFFER
CC, UC, LC, IC, TRIM, MCC
```

In the future, more options will be added to this list.

An option that is set for a specific unit will always be used for that unit regardless of the general setting of the option, that is, the default setting or a setting made by calling FREADC with specifying the "unit" parameter.

The special options that are available for FREADC are as follows:

BUFFER	LENGTH	PREFIX
CONTINUATION	LINENUMBER	REREAD
DELIMITERS	LONG	RESET
ECHO	MTSLNR	RESTORE
ENDFILE	NAMES	SAVE
ENDLINE	NOFDUB	SHORT
ERROR	NOFILL	STREAM
EVEN	NULL	TYPE
INFORMATION	NUMBER	UPDATE
JUSTIFY	ORMTS	VERBOSITY
LASTDELIMITER	QUOTE	

Special Output Options for FWRITE

Three special output options may be used with FWRITE. To set an output option, either of the following statements may be used:

```
CALL FWRITC('option=value;')

CALL FWRITC('option=?;',value)
```

where "option" specifies the option to be used and "value" specifies the value of the option. The first argument must end with ";". Legal values for logical options are ON (or TRUE), and OFF (or FALSE). Otherwise, values are as specified in the description of the option.

To reset any of the output options, use the keyword 'DEFAULT'. For example,

```
CALL FWRITC('ORL=DEFAULT;')
```

will set the global output record length to 76.

Output options can be set for a specific unit only by specifying either of the following statements:

```
CALL FWRITC(unit,'option=value;')

CALL FWRITC(unit,'option=?;',value)
```

Control Printer Spacing (CC)

The CC option is used to control printer spacing. By default, all output lines are written with carriage control. Line spacing is controlled via the slash "/" operator in the message.

To make use of the allowable printer spacing controls, specify CC=ON. If this option is used, a legal carriage control character must be specified as the first character of every output line. Printer carriage control is described in Appendix H to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System.

For example:

```
CALL FWRITC(6,'CC=ON;')
```

Indexed Output (LINENUMBER)

The LINENUMBER (or LINE) option is used to do indexed output. To write a record to a specified line number, specify either

```
CALL FWRITC([unit,]'LINE=lnr;')

CALL FWRITC([unit,]'LINE=?;',lnr)
```

The value specified by "lnr" is the internal form of the line number, i.e., the line number times 1000. Using this option causes the next record written via FWRITE to be written indexed using the value in "lnr". Subsequent records will be written sequentially. Restriction: FWRITE should be called immediately after FWRITC before any calls to FREAD are made since reading a record will change the MTS line number.

Maximum Output Record Length (ORL)

The ORL option is used to set the maximum output record length of records written. Phrases that are longer than the ORL are broken at the nearest blank and continued on the following record. The default length for terminals is 76 characters. To change this value, specify either

```
CALL FWRITC([unit,]'ORL=n;')

CALL FWRITC([unit,]'ORL=?;',n)
```

where "n" is a value from 20 to 255, and defines the maximum number of characters in a line of output.

BUFFER

Prototype: CALL FREADC([unit,]'BUFFER;',name)

where "name" is the name of an array, usually 260 bytes in length.

Action: To have FREAD return the current input buffer (see the section "Reading from a User-Supplied Buffer").

Notes: The array should be dimensioned at least as long as LRECL+4 bytes, where LRECL is the longest input record length. Keep in mind that LRECL can be increased by error recovery editing; this is why the array should usually be 260 bytes in length. The buffer index is returned as 0 when the length is 0.

The unit may be specified to return the input buffer for that unit. If the unit is not specified, the last unit from the FREAD call will be used.

Example:
```
INTEGER*2 IBUFF(130)
CALL FREADC(5,'BUFFER;',IBUFF)

     then IBUFF(1) = buffer length in bytes
          IBUFF(2) = buffer index
          IBUFF(3),IBUFF(4),...
                   = buffer proper
```

returns the buffer from the last call of FREAD with logical unit 5.

CC

Prototypes:
```
CALL FWRITC([unit,]'CC={ON|OFF};')
CALL FWRITC([unit,]'CC=?',{.TRUE.|.FALSE.})
```

Action: The CC option controls carriage-control. If ON, the first character of every output line is interpreted as a carriage-control character. The list of legal carriage-control characters is given in Appendix H to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System. If the unit is specified, the option will be applied to that unit only; otherwise, it will be applied to all FWRITC calls.

Default: The default for the CC option depends on the setting of the MTS @CC FDname modifier (which defaults ON).

Example:
```
CALL FWRITC(6,'CC=ON;')
```

CONTINUATION

Prototypes: CALL FREADC('CONTINUATION=''string'';')
CALL FREADC('CONTINUATION;','string')

where "string" is a string of weak continuation characters followed by a string of strong continuation characters, the strings themselves being enclosed and separated by any delimiter.

Action: To define a new set of continuation characters for FREAD to use. A weak continuation character appearing as the last nonblank character on an input line will cause a new line to be read only if more variables require data. A strong continuation character appearing as the last nonblank character on an input line always causes a new line to be read.

Default: No continuation characters are defined.

Example: CALL FREADC('CONT=''/-//'';')

defines the minus sign (-) to be a weak continuation character.

DELIMITERS

Prototypes: CALL FREADC('DELIMITERS=''string'';')
CALL FREADC('DELIMITERS;','string')

where "string" is a string of weak delimiter characters followed by a string of strong delimiter characters, the strings themselves being enclosed and separated by a nondelimiter.

Action: To define a new set of delimiters for FREAD to use. The only difference between weak and strong delimiters is that a strong delimiter appearing as the last nonblank character on a line will terminate stream input, whereas a weak delimiter will not.

Default: Originally, two weak delimiters are defined (blank and comma), and no strong delimiters are defined.

Notes: Blank may not be defined as a strong delimiter. Observe that '1. 2' is interpreted as two numbers when blank is a delimiter, and as one number ('1.02') when it is not.

Examples: CALL FREADC('DELIMITERS=''/, /;/'';')

defines comma and blank as weak delimiters, and the semicolon as a strong delimiter. The slash is both enclosing and separating the delimiter strings.

CALL FREADC('DELI=DEFAULT;')

resets the delimiters (that is, blank and comma are weak delimiters, not strong delimiters).

ECHO

Prototypes: CALL FREADC('ECHO={ON|OFF};')
CALL FREADC('ECHO=?;',{.TRUE.|.FALSE.})

CALL FWRITC('ECHO={ON|OFF};')
CALL FWRITC('ECHO=?;',{.TRUE.|.FALSE.})

Action: If FREADC is called with this option ON, all lines read by FREAD from files and nonterminal devices are echoed on SERCOM.

If FWRITC is called with this option ON, all lines written by FWRITE are echoed on SERCOM.

Default: OFF

Example: CALL FREADC('ECHO=ON;')

ENDFILE

Prototypes: CALL FREADC('ENDFILE={0...3|MTS};')
CALL FREADC('ENDFILE=?;',{0...3|'MTS'})

Action: To define what happens when an end-of-file condition is encountered on a read. If 0, 1, 2, or 3, then FREAD will fill unsatisfied variables with zeros or blanks (character strings are filled with blanks, all other types are filled with zeros) and RETURN 1 through 3 to the calling program. If MTS, then FREAD will drop into MTS command mode so that, for example, I/O units may be reassigned.

Default: Drop into MTS command mode; a $RESTART command will cause the reading of another data line from the same unit.

Note: The filling of unsatisfied variables with zeros or blanks may be suppressed using the NOFILL option.

Example: CALL FREADC('ENDFILE=2;')

causes a RETURN 2 exit to be taken to the program that called FREAD whenever an end-of-file is encountered.

ENDLINE

Prototypes: CALL FREADC('ENDLINE={0...3|STREAM};')
CALL FREADC('ENDLINE;',{0...3|'STREAM'})

Action: To define what happens when an end-of-line condition is encountered (that is, the user requests more data fields to be read than there are fields left on a line). If 0, 1, 2, or 3, then FREAD will fill unsatisfied variables with zeros or blanks and RETURN 0 through 3 to the calling program. If STREAM, then stream input is enabled.

Default: Unsatisfied variables are filled with zeros or blanks (character strings are filled with blanks, all other types are filled with zeros), and a RETURN 1 exit is taken to the program that called FREAD.

Notes: When stream input is terminated by a strong delimiter, FREAD will RETURN 0 to the calling program. The filling of unsatisfied variables with zeros or blanks may be suppressed using the NOFILL option.

"Stream input" can also be controlled by using the STREAM option.

Example: CALL FREADC('ENDLINE=STREAM;')

enables stream input. As many lines as are necessary will be read until all variables are satisfied, or until a strong delimiter is encountered as the last nonblank character of a line, or until an end-of-file condition is encountered.

ERROR

Prototypes: CALL FREADC('ERROR={0...3|MTS|RECOVER|SIGNOFF|LINE|
FIELD};')
CALL FREADC('ERROR;',{0...3|'MTS'|'RECOVER'|'SIGNOFF'|
'LINE'|'FIELD'})

Action: To define what happens when a conversion error occurs. If 0,...,3 then when a conversion error occurs, FREAD

will RETURN 0 through 3 to the calling program. If MTS, then when a conversion error occurs, FREAD will return to MTS command mode (a $RESTART will result in a prompt for replacement data in conversational mode). If RECOVER, then FREAD will attempt conversational error recovery when a conversion error occurs. This means that the user will be prompted for either a replacement field, or for the rest of the line from the point of error. If SIGNOFF, then FREAD will sign the user off if a conversion error occurs. LINE is much like RECOVER except that the user is only prompted for the rest of the line from the point of the error when a conversion error occurs. FIELD is also much like RECOVER except that the user is only prompted for a replacement field when a conversion error occurs.

Default: RECOVER is the default in conversational mode, and SIGNOFF is the default in batch mode.

Note: The setting of the ERROR option is independent of the verbosity of error messages which is controlled by the VERBOSITY option.

Example:
```
CALL FREADC('ERROR=2;')
```

causes a RETURN 2 exit to be taken to the calling program when a conversion error occurs.

EVEN

Prototypes:
```
CALL FREADC('EVEN={ON|OFF};')
CALL FREADC('EVEN;',{.TRUE.|.FALSE.})
```

Action: If ON, then hexadecimal strings with an odd number of digits are recognized as errors.

Default: Hexadecimal strings may have an even or odd number of digits.

Example:
```
CALL FREADC('EVEN=ON;')
```

IC

Prototypes:
```
CALL FREADC([unit,]'IC={ON|OFF};')
CALL FREADC([unit,]'IC=?',{.TRUE.|.FALSE.})
```

Action: The IC option controls implicit concatenation. If ON, implicit concatenation is enabled which means that "$CONTINUE WITH" lines are recognized for implicit concatenation. If OFF, implicit concatenation is disabled.

Default: The default for the IC option depends on the settings of the option $SET IC command (which defaults ON) and the MTS @IC FDname modifier (which defaults ON).

Example:
```
CALL FREADC('SCARDS','IC=OFF;')
```

INFORMATION

Prototype:
```
CALL FREADC('INFORMATION;',array)
```

where "array" is a four-element INTEGER*2 array.

Action: To return information about a conversion error.

Note: This option is one which facilitates user program conversion error processing. See also the LASTDELIMITER option, the TYPE option, and the BUFFER option.

Example:
```
INTEGER*2 CODES(4)
CALL FREADC('INFO;',CODES)
```

then CODES(1) = error code (see below)
CODES(2) = starting column of field in error
CODES(3) = field width
CODES(4) = column of character in error
(if apropos, otherwise 0)

The error code will be one of:

0 - no error occurred
1 - real, syntax
2 - real, illegal character
3 - real, machine limits
11 - integer, syntax
12 - integer, illegal character
13 - integer, machine limits
21 - character string, too short
22 - character string, too long
23 - character string, missing closing quote
24 - character string, not enclosed in quotes
31 - hex string, too short
32 - hex string, too long
33 - hex string, illegal hex digit
34 - hex string, odd number of digits
41 - binary string, too short
42 - binary string, too long
43 - binary string, illegal binary digit
51 - octal string, too short
52 - octal string, too long
53 - octal string, illegal octal digit
61 - line number, too many digits before decimal

62 - line number, too many digits after decimal
63 - line number, syntax
64 - line number, illegal character

JUSTIFY

Prototypes: CALL FREADC('JUSTIFY={RIGHT|LEFT};')
CALL FREADC('JUSTIFY;',{'RIGHT'|'LEFT'})

Action: Subsequent string conversions will be either right- or left-justified.

Default: Character strings are left-justified, and truncated or padded with blanks on the right. Hexadecimal, octal, and binary strings are right-justified, and truncated or padded with zeros on the left.

Example: CALL FREADC('JUSTIFY=LEFT;')

causes subsequent string conversions to be left-justified.

LASTDELIMITER

Prototype: CALL FREADC('LASTDELIMITER;',var)

where "var" is a LOGICAL*1 variable.

Action: To return the delimiter that followed the last input data field. One character is returned.

Note: Initially, and at the start of each input line, it is set to hexadecimal zero (X'00'). X'FF' means an end-of-line delimited the last field.

Example: CALL FREADC('LAST;',ICHAR)

LC

Prototypes: CALL FREADC([unit,]'LC={ON|OFF};')
CALL FREADC([unit,]'LC=?',{.TRUE.|.FALSE.})

CALL FWRITC([unit,]'LC={ON|OFF};')
CALL FWRITC([unit,]'LC=?',{.TRUE.|.FALSE.})

Action: The LC option controls uppercase conversion. If ON, input or output lines are not converted to all uppercase

(the lines remain unchanged). If OFF, input or output lines are converted to all uppercase.

Default: The default is ON unless overridden by the setting of the MTS @LC/UC FDname modifier (which defaults to LC).

Example:
```
CALL FREADC(5,'LC=OFF;')
```

converts all input lines read from unit 5 to uppercase.

LENGTH

Prototypes:
```
CALL FREADC('LENGTH={ON|OFF};')
CALL FREADC('LENGTH;',{.TRUE.|.FALSE.})
```

Action: If ON, then the actual length of a string found is returned in the parameter describing the length of the string buffer.

Default: The length of strings found is not returned.

Notes: The length returned is not necessarily the length in bytes, but rather the actual number of characters or digits found. The length returned is the length prior to any padding or truncation. It is important that if LENGTH is set to ON, then the parameter describing the user-buffer length must be a variable and not a constant.

Example:
```
CALL FREADC('LENGTH=ON;')
LEN=4
CALL FREAD(5,'HEX:',ADDR,LEN)
```

returns the actual number of hexadecimal digits found in LEN.

LINENUMBER

Prototypes:
```
CALL FREADC([unit,]'LINENUMBER=lnr;')
CALL FREADC([unit,]'LINENUMBER=?;',lnr)

CALL FWRITC([unit,]'LINENUMBER=lnr;')
CALL FWRITC([unit,]'LINENUMBER=?;',lnr)
```

where "lnr" is a fullword integer MTS line number in internal form (this is equivalent to the external form * 1000; e.g., line number 1 is expressed as 1000).

Action: The next line read by FREAD will be indexed at the given line number. Subsequent reads will be sequential.

Default: All reads are done sequentially.

Note: This action is independent of error recovery reads which always occur on GUSER.

Examples:
```
CALL FREADC(5,'LINE=2000;')
CALL FREAD(5,'I:',NUMBER)
```

requests FREAD to read an integer from line 2.000 on I/O unit 5.

```
CALL FREADC('LINE=DEFAULT;')
```

causes subsequent lines to be read sequentially.

LONG

Prototypes:
```
CALL FREADC('LONG={ON|OFF};')
CALL FREADC('LONG;',{.TRUE.|.FALSE.})
```

Action: If ON, then strings that are too long are recognized as errors.

Default: Strings that are too long are truncated.

Example:
```
CALL FREADC('LONG=ON;')
```

MCC

Prototypes:
```
CALL FWRITC([unit,]'MCC={ON|OFF};')
CALL FWRITC([unit,]'MCC=?',{.TRUE.|.FALSE.})
```

Action: The MCC option controls machine carriage-control. If ON, the first character of every output line is interpreted as a machine carriage-control character. The list of legal machine carriage-control characters is given in Appendix H to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System. If the unit is specified, the option will be applied to that unit only; otherwise, it will be applied to all FWRITC calls.

The use of machine carriage-control characters is device-dependent and not recommended.

Default: The default for the MCC option depends on the setting of the MTS @MCC FDname modifier (which defaults OFF).

MTSLNR

Prototype: CALL FREADC([unit,]'MTSLNR;',var)

where "var" is an INTEGER*4 variable.

Action: To return the MTS line number of the last line read by FREAD (this is the internal form of the MTS line number which is equivalent to the external form * 1000, e.g., line number 1 is returned as 1000).

Example: CALL FREADC('MTSLNR;',LNR)

NAMES

Prototypes: CALL FREADC('NAMES=''names'';')
CALL FREADC('NAMES;','names')

where "names" is a list of ten names, each enclosed and separated by a delimiter (see the example below).

Action: When a conversion error is encountered and the verbosity level is at least one, the first action FREAD takes is to print a diagnostic of the form:

```
?INVALID x:  "y"
```

where the actual data field in error is substituted for "y", and the name of the data type is substituted for "x". By default, the name of the data type is one of the following:

```
1  - character string
2  - hexadecimal string
3  - binary string
4  - octal string
5  - integer number
6  - real number
7  - logical number
8  - MTS line number
```

By using the NAMES option the user is able to define eight names which will be substituted for "x", in place of the eight default names given above. Such is the function of the first eight names which the user must provide.

When a conversion error is encountered and FREAD is allowed to prompt for replacement data (which is the default strategy), then the prompt printed by FREAD is of the form:

```
?ENTER REPLACEMENT z, OR RE-ENTER ...
```

where "z" is either STRING or NUMBER, depending on whether the data in error was of string or numeric type. By using the NAMES option the user is able to define two names of his own which will be substituted for "z", in place of those just listed. Such is the function of the last two names that the user must provide.

Notes: Each name must be no longer than sixteen characters. A null name (that is, a name of length zero) is interpreted to mean that the corresponding name is not to be redefined. Trailing blanks are always trimmed from names before they are printed.

Example:
```
CALL FREADC('NAMES=''///// /////RATE/'';')
CALL FREADC('ORMTS=OFF;')
CALL FREADC('ERROR=FIELD;')
CALL FREADC('VERBOSITY=1;')
```

then an invalid integer (123:) would produce the following response from FREAD:

```
?INVALID:  "123:"
?ENTER REPLACEMENT RATE
?
```

NOFDUB

Prototypes:
```
CALL FREADC('NOFDUB={ON|OFF};')
CALL FREADC('NOFDUB;',{.TRUE.|.FALSE.})
```

Action: To specify that the "I/O unit" on a subsequent call to FREAD or FWRITE is not a FDUB.

Normally, FREAD and FWRITE call the system subroutine CHKFDUB to differentiate between FDUBs and user-supplied buffers. By setting the NOFDUB switch to ON, the user can specify that the unit is actually a buffer and avoid the overhead of calling CHKFDUB. However, it is preferable to call either FREADB or FWRITB.

Default: OFF

Example:
```
CALL FREADC('NOFDUB=ON;')
```

NOFILL

Prototypes: CALL FREADC('NOFILL={ON|OFF};')
CALL FREADC('NOFILL;',{.TRUE.|.FALSE.})

Action: If OFF, then unsatisfied variables are filled with zeros or blanks when an end-of-file condition or an end-of-line condition occurs which will result in a return to the user program.

Default: OFF

Example: CALL FREADC('NOFILL=ON;')

NULL

Prototypes: CALL FREADC('NULL={ON|OFF};')
CALL FREADC('NULL;',{.TRUE.|.FALSE.})

Action: If ON, then variables corresponding to null fields are filled with zeros or blanks.

Default: Variables corresponding to null fields are left unchanged.

Note: A null field is defined as either a delimiter appearing as the first nonblank character on a line, or as two nonblank delimiters with no nonblank characters between them.

Example: CALL FREADC('NULL=ON;')

NUMBER

Prototype: CALL FREADC('NUMBER;',var)

where "var" is an INTEGER*4 variable.

Action: To return the number of fields found on the last read call to FREAD. This count includes only those fields successfully processed.

Note: The number of fields is also returned by FREAD as a real and an integer FUNCTION value.

Example: NUMFLD=FREAD(5,'R...',R1,R2,R3,R4)

or

```
CALL FREAD(5,'R...',R1,R2,R3,R4)
CALL FREADC('NUMBER;',NUMFLD)
```

NUMFLD will be equal to the number of fields found.

ORL

Prototypes:
```
CALL FWRITC([unit,]'ORL=n;')
CALL FWRITC([unit,]'ORL=?',n)
```

Action: The ORL option sets the maximum output record length of lines written. Phrases longer than the ORL specification will be broken at the nearest blank and continued on the following line.

Default: The length is set to 76.

Example:
```
CALL FWRITC(6,'ORL=132;')
```

sets the maximum output record length for unit 6 to 132.

ORMTS

Prototypes:
```
CALL FREADC('ORMTS={ON|OFF};')
CALL FREADC('ORMTS;',{.TRUE.|.FALSE.})
```

Action: If OFF, then the string 'OR "MTS"' will not be appended to prompt for replacement data, and FREAD will not interpret the string "MTS" as meaning return to MTS command mode.

Default: ON

Example:
```
CALL FREADC('ORMTS=OFF;')
```

PREFIX

Prototypes:
```
CALL FREADC('PREFIX=''char'';')
CALL FREADC('PREFIX;','char')
```

where "char" is any single character.

Action: To define a new FREAD prefix character. Every data line read and every error message line written by FREAD is prefixed with this character.

Default: '?'

Example: CALL FREADC('PREFIX=''-'';')

sets the FREAD prefix character to "-".

QUOTE

Prototypes: CALL FREADC('QUOTE={ON|OFF};')
CALL FREADC('QUOTE;',{.TRUE.|.FALSE.})

Action: If ON, then character strings must be enclosed within quotes (e.g., 'string'). If OFF, then quotes around character strings are not recognized as having any special significance (that is, they are considered as part of the string rather than delimiting the string).

Default: Character strings may or may not be enclosed in quotes.

Note: Quotes around character strings are a necessity when a delimiter is embedded within the string.

Example: CALL FREADC('QUOTE=OFF;')

disables the recognition of quotes as character string delimiters.

REREAD

Prototypes: CALL FREADC('REREAD={ON|OFF};')
CALL FREADC('REREAD;',{.TRUE.|.FALSE.})

Action: To enable or disable the REREAD option. If ON, then input buffer pointers will be set up to reread the field if a conversion error has occurred. FREAD will return to the user program after the conversion error in accordance with the ERROR option. On the next call to FREAD, the field in error will be read again.

Default: OFF, normally the input buffer pointers remain positioned to read the next input data field.

Example: CALL FREADC('REREAD=ON;')

RESET

Prototype: CALL FREADC('RESET;')

Action: Resets all FREAD options to what they were initially. Stacked options are not lost.

Example: CALL FREADC('RESET;')

RESTORE

Prototype: CALL FREADC('RESTORE;'[,var])

where "var" is an INTEGER*4 variable (optional).

Action: To restore all FREAD options to what they were on the previous SAVE. If the second argument is provided, the stack level at which the options were restored is returned in it. This is useful only for checking that options are restored from the same stack level at which they were saved.

Notes: The number of restores must never exceed the number of saves.

Example:
```
C** SAVE OPTIONS BEFORE CALL
        CALL FREADC('SAVE;',LVL)
        CALL SUBPGM
C** RESTORE OPTIONS AFTER CALL
        CALL FREADC('REST;',LVL1)
C** CHECK STACK LEVEL
        IF(LVL .NE. LVL1) GOTO 999
```

SAVE

Prototype: CALL FREADC('SAVE;'[,var])

where "var" is an INTEGER*4 variable (optional).

Action: To save the current status of all FREAD options, including the delimiters. They are stacked automatically in an area internal to FREAD. A subsequent RESTORE will restore the options to exactly as they were at the time of the last SAVE. If the second argument is provided, the stack level at which the options were saved is returned in it. This is useful only for checking that options are restored from the same stack level at which they were saved.

Notes: Stacking options allows FREAD to be used independently from within different subprograms.

If a call to FREADC with "SAVE;" is made, the stack level will not be returned.

Examples:
```
CALL FREADC('SAVE;')
CALL FREADC('SAVE;',LEVEL)
```

SHORT

Prototypes:
```
CALL FREADC('SHORT={ON|OFF};')
CALL FREADC('SHORT;',{.TRUE.|.FALSE.})
```

Action: If ON, then strings that are too short are recognized as errors.

Default: Strings that are too short are padded with zeros or blanks.

Example:
```
CALL FREADC('SHORT=ON;')
CALL FREADC('LONG=ON;')
```

Forces all strings that do not fit exactly into the user-supplied input buffer to be treated as conversion errors.

STREAM

Prototypes:
```
CALL FREADC('STREAM={ON|OFF};')
CALL FREADC('STREAM;',{.TRUE.|.FALSE.})
```

Action: To enable (ON) or disable (OFF) stream input.

Default: OFF, stream input is disabled.

Note: Stream input can also be controlled by using the ENDLINE option.

Example:
```
CALL FREADC('STREAM=ON;')
```

TRIM

Prototypes:
```
CALL FREADC([unit,]'TRIM={ON|OFF};')
CALL FREADC([unit,]'TRIM=?',{.TRUE.|.FALSE.})

CALL FWRITC([unit,]'TRIM={ON|OFF};')
CALL FWRITC([unit,]'TRIM=?',{.TRUE.|.FALSE.})
```

Action: The TRIM option controls the trimming of trailing blanks from input or output records. If ON, all trailing blanks except one are removed from the line. If OFF, the line is not changed.

Default: The default is OFF unless the MTS @TRIM FDname modifier is explicitly specified for the logical I/O unit in which case the default is ON.

Example:
```
CALL FREADC(4,'TRIM=OFF;')
```

sets the TRIM option to OFF for unit 4.

TYPE

Prototype:
```
CALL FREADC('TYPE;',var)
```

where "var" is an INTEGER*4 variable.

Action: To return the type code corresponding to the last data field processed. It will be one of the following:

0 - initially
1 - character string
2 - hexadecimal string
3 - binary string
4 - octal string
5 - integer number
6 - real number
7 - logical number
8 - MTS line number

Example:
```
CALL FREADC('TYPE;',ITYPE)
```

UC

Prototypes:
```
CALL FREADC([unit,]'UC={ON|OFF};')
CALL FREADC([unit,]'UC=?',{.TRUE.|.FALSE.})

CALL FWRITC([unit,]'UC={ON|OFF};')
CALL FWRITC([unit,]'UC=?',{.TRUE.|.FALSE.})
```

Action: The UC option controls uppercase conversion. If ON, input or output lines are converted to all uppercase. If OFF, input or output lines are not converted (the lines remain unchanged).

Default: The default is OFF unless overridden by the setting of the MTS @LC/UC FDname modifier (which defaults to LC).

Example:
```
CALL FREADC(5,'UC=ON;')
```
converts all input lines read from unit 5 to uppercase.

UPDATE

Prototypes:
```
CALL FREADC('UPDATE={ON|OFF};')
CALL FREADC('UPDATE;',{.TRUE.|.FALSE.})
```

Action: If ON, FREAD will automatically update the buffer index to correspond to the next data field when reading from a user-provided buffer.

Default: OFF, FREAD does not change the buffer index.

Note: When the updated buffer index exceeds the buffer length, there are no more data fields left in the buffer. The user program should check for this condition (see the section "Reading from a User-Supplied Buffer").

Example:
```
CALL FREADC('UPDATE=ON;')
```

VERBOSITY

Prototypes:
```
CALL FREADC('VERBOSITY={0...3};')
CALL FREADC('VERBOSITY;',{0...3})
```

Action: Controls the verbosity of error messages generated by FREAD in response to conversion errors. 0 is the least verbose (prints nothing) and 3 is the most verbose.

Default: The default verbosity is 2 for conversational mode, and 3 for batch mode.

Note: Control over the text of error messages is provided by the ORMTS option and the NAMES option.

Example: The following is a typical verbose error message,

```
(a) ?INVALID INTEGER NUMBER: "-2-"
(b) ?SYNTAX                    $
(c) ?UNIT=5, FDNAME=-FILE AT LINE 13.07
```

Verbosity Level	Lines Printed
0	none
1	(a)
2	(a) and (b)
3	(a),(b), and (c)

October 1983

ADVANCED USES OF FREAD AND FWRITE

Reading Arrays

To read data into an "n x m" array it is essential that the user understand how FORTRAN stores data in these arrays. Consider the array X(3,3). X(1,1) is the first element, X(2,1) is the second element and so on. In other words, FORTRAN stores its arrays such that the first subscript varies the quickest. In fact, the array could be thought of as 3 vectors starting at X(1,1), X(1,2), and X(1,3). Thus, if the data is in that order, FREAD can be used to read it:

```
CALL FREAD(unit,'R V:',X(1,1),3)
CALL FREAD('*','R V:',X(1,2),3)
CALL FREAD('*','R V:',X(1,3),3)
```

The following FORTRAN read is equivalent:

```
     READ(5,100) (X(I,J),I=1,3),J=1,3)
100  FORMAT(9F10.2)
```

Indexed Input and Output

There is always that one time when a record must be read from a particular line or write one to a particular line. This means INDEXED I/O must be performed. To do this with FREAD and FWRITE requires use of their buffering capabilities. To read one or more variables from line 3.000, the LINENUMBER option must be used as follows:

```
CALL FREADC('LINENUMBER=3000;')
CALL FREAD(5,'I V:',INDEX,N)
```

Note that the form of the line number in the option call is in internal form (i.e., line number * 1000). Any further read is done sequentially, i.e., the LINENUMBER option is a one-shot option.

The following commands illustrate how to write a line or record to a specified location in a file:

```
CALL FWRITC('LINE=3000;')
CALL FWRITE(8,'This is written indexed at line 3.000: ')
```

Note that the line number is specified in its internal form (i.e., line number * 1000).

Using FREAD Without Transferring Data

It is possible to direct FREAD to read a line (or a user-supplied buffer), and return to the calling program without transferring any data. This is done as follows:

```
CALL FREAD(5,':',DUMMY)
```

The above call would cause a line to be read from logical I/O unit 5. The second argument is the type string, which in this case does not contain any type codes, hence FREAD will return immediately. The third argument, DUMMY, must be supplied for reasons internal to FREAD.

The advantage attained by having FREAD read a line without processing the data on the line is that it simplifies some of the cases where FREAD is commonly used. For example, consider the following program that reads and adds integer numbers from SCARDS until an end-of-file is encountered, and then prints the sum of the numbers. Any number of integers may be on a given line, and invalid integers are not included in the sum.

```
      CALL FREADC('ENDFILE=2;')
      CALL FREADC('ERROR=3;')
      ISUM=0
1     CALL FREAD('SCARDS',':',DUMMY,&1,&3,&1)
2     CALL FREAD('*','I:',I,&1,&3,&2)
      ISUM=ISUM+I
      GO TO 2
3     WRITE (6,4) ISUM
4     FORMAT('ISUM=',I10)
      STOP
      END
```

The following program is much like the program of the above example, except that it sums integer and real numbers. A given number is first examined to see if it is a valid integer number. Failing that, it is examined to see if it is a valid real number, and failing that, if neither integer nor real, it will not be included in either of the sums. Note, in particular, the use of the REREAD option.

```
      CALL FREADC('ERROR=3;')
      CALL FREADC('ENDFILE=2;')
      ISUM=0
      RSUM=0.0
1     CALL FREAD('SCARDS',':',DUMMY,&1,&4,&1)
2     CALL FREADC('REREAD=ON;')
      CALL FREAD('*','I:',I,&1,&4,&6)
      ISUM=ISUM+I
      GO TO 2
4     WRITE(6,5) ISUM,RSUM
5     FORMAT(' ISUM= ',I12/' RSUM= ',G15.7)
      STOP 2
```

```
6     CALL FREADC('REREAD=OFF;')
      CALL FREAD('*','R:',R,&1,&4,&3)
      RSUM=RSUM+R
      GO TO 2
      END
```

Input Subroutines

There are times when the user may want to use one input routine for several applications. However, options which are required in the subroutine may conflict with those in the calling program. The SAVE and RESTORE options are used to solve this problem:

```
C     Main program
       ...
      CALL INPUT(....)
       ...
      END
      SUBROUTINE INPUT(....)
      CALL FREADC('SAVE;')
C     Set options for the subroutine
      CALL FREAD( .... )
C     Perform the necessary calls to read data
       ...
C     Before returning, restore caller's options
      CALL FREADC('RESTORE;')
      RETURN
      END
```

Creating Text Lines For Other Routines

Sometimes it is necessary to use a subroutine that requires a parameter that is part text and part numbers. If the user does not want the number part to be variable, it is not easy to do in FORTRAN. However, using FWRITB it is very simple. First, use FWRITB to create the required string with the correct text and values. Then use the resulting buffer and length to call the subroutine. A common example is using the CNTRL subroutine to set some file or tape parameter:

```
      INTEGER*2 CNTRLL/32/
      INTEGER BLKSIZ/4000/,BLKLEN/2/
      LOGICAL*1 BLKTYP(2)/'F','B'/,CNTRLP(32)
      LRECL = 80
      CALL FWRITB(CNTRLP,CNTRLL,'FMT=<S>(; ',BLKTYP,BLKLEN)
      CALL FWRITB(CNTRLP,CNTRLL,'<I>,<I>): ',BLKSIZ,LRECL)
C   CNTRLP will now contain 'FMT=FB(4000,80)'.
      CALL CNTRL (CNTRLP, CNTRLL, TAPE)
```

FREAD/FWRITE EXAMPLES

The following example shows how one might use FREAD and FWRITE to read and write titles.

```
      LOGICAL*1 TITLE(132)
      INTEGER   TAB, PAGNO, TITLEN
      PAGNO = 0
C  Set up output options for unit 6.
      CALL FWRITC(6,'ORL=132;')
      CALL FWRITC(6,'CC=ON;')
C  Read in title from unit 5. Set max length of title.
      TITLEN = 132
      CALL FREAD(5,'M:',TITLE,TITLEN)
C  TITLEN now has true length of title. TITLE has data.
      ...
C  Now update page number for new page and write title.
C  Also center the title on the page.
      PAGNO = PAGNO + 1
      TAB = (132 - TITLEN ) / 2
      CALL FWRITE(6,'1<T><STRING,T125>Page <I>: ',
     1             TAB,TITLE,TITLEN,PAGNO)
      ...
```

This method of reading and writing characters uses far less CPU time than FORTRAN's nA1 format.

The following example uses various options to recover from a known error in the data file. In this case, it was known that some 'E's were incorrectly punched as 'F's.

```
C
C  An example of sneaky error recovery. Illegal character
C  errors are assumed to be 'E's punched as 'F's.
C
      INTEGER*2 IBUFF(130),CODES(4)
      LOGICAL*1 BUFF(256)
      INTEGER   FLDCNT,ERRCNT
      REAL      X(10)
C
      EQUIVALENCE (BUFF(1),IBUFF(3))
C
C  Initialize variables and FREAD
C
      ERRCNT = 0
      CALL FREADC('ENDFILE=2;')
      CALL FREADC('ERROR=3;')
C  Set error message printing off because we know they
C  exist but we don't want the output cluttered up.
      CALL FREADC('VERBOSITY=0;')
 10   CALL FREAD(5,'REAL VECTOR: ',X,10,&11,&20,&30)
 11   CONTINUE
```

```
C
C  Process data.
C
      GO TO 10
C
C  End-of-file indicates end of program. Print error count
C  and stop.
C
 20   CALL FWRITE(6,' <I> NUMBERS IN ERROR: ',ERRCNT)
      STOP
C
C  Error detected. Get input buffer, number of fields read,
C  information about the error. If error not the type
C  expected, ignore it.
C
 30   CALL FREADC('BUFFER;',IBUFF)
      CALL FREADC('NUMBER;',FLDCNT)
      CALL FREADC('INFORMATION;',CODES)
      IF (CODES(1) .NE. 2) GO TO 40
      ERRCNT = ERRCNT + 1
C
C  Reconstruct the record by replacing the 'F' with an 'E'.
C  Must reset length, field pointer, and read via BUFFER
C
 35   IBUFF(2) = CODES(2)
      K = CODES(4)
      CALL MOVEC(1,'E',BUFF(K))
C  Recalculate the number of fields left to read.
      N = 10 - FLDCNT + 1
      CALL FREADB(IBUFF,'REAL VECTOR:',X(FLDCNT),N,&11,&20,&30)
      GO TO 11
C
C  Unexpected error...print record and ignore it.
C
 40   CALL FWRITE(6,' Unexpected illegal input record /; ')
      I = IBUFF(1)
      CALL FWRITE(6,' <STRING>; ',IBUFF(3),I)
      GO TO 10
      END
```

DATA DESCRIPTIONS

Integer Numbers

An integer number consists of a signed or unsigned decimal digit string without a decimal point. Multiple signs are accepted and evaluated using the usual algebraic rules, e.g., --=+. The number must fall within the range of -2147483648 to 2147483647. Range checking appropriate to INTEGER*2 conversion is not performed, so that INTEGER*2 conversion is equivalent to INTEGER*4 conversion followed by an assignment statement.

Real Numbers

A real number consists of a signed or unsigned decimal digit string optionally containing a decimal point, and optionally followed by an exponent field. Multiple signs are always evaluated, whether in the fraction or the exponent. An exponent field consists of an exponent starter (E or D) optionally followed by an integer number. Real numbers must fall in the range of .539760534693402789D-78 to .72370055773322621 1D+76.

The following are examples of invalid real numbers:

1-	sign character preceded by digit
.-2	sign character following decimal point without an intervening exponent starter (E or D)
3E-	exponent sign character the last character in field
4..	decimal point after decimal point
5E.	decimal point after exponent starter
6DD	exponent starter after exponent starter
-E7	exponent starter immediately follows sign character

Logical Numbers

Any characters, except the current delimiters, are valid in a logical number. The field is scanned and the first T or F encountered determines whether the number is .TRUE. or .FALSE.; if neither is found, .FALSE. is assumed. Internally, .FALSE. is integer 0 and .TRUE. is integer 1.

MTS Line Numbers

An MTS line number is a string of the form "snnnnn.nnn" where "s" is the sign (+ or -) and "n" is a decimal digit. Leading '+' signs and zeros, trailing decimal points and trailing zeros after the decimal may be omitted. Internally, the MTS line number is converted to an INTEGER*4 value multiplied by 1000; e.g., line number 1 is converted to 1000.

Character Strings

A character string is a sequence of any characters. One character occupies one byte of storage. By default:

(1) Character strings may or may not be enclosed in single quotes ('). Normally, a character string does not need to be enclosed in quotes unless one or more of the characters within the string is defined as a delimiter. If a string is enclosed in quotes, then a quote within the string must be represented by two successive quotes.
(2) Character strings are left-justified to the byte.
(3) Character strings that are shorter than requested are padded on the right with blanks.
(4) Character strings that are longer than requested are truncated on the right.

Optionally:

(1) By setting the QUOTE option appropriately, character strings that are not enclosed in quotes are treated as errors. Another setting of the QUOTE option causes quotes enclosing character strings not to be recognized as having any special significance. That is, the quotes are regarded as part of the string text proper.
(2) By setting the JUSTIFY option to RIGHT, character strings will be right-justified. Right-justification implies that padding and truncation occur on the left.
(3) By setting the SHORT option to ON, character strings that are too short will be treated as errors.
(4) By setting the LONG option to ON, character strings that are too long will be treated as errors.

Hexadecimal Strings

A hexadecimal string is a sequence from the set of characters 0 through 9 and A through F. Two hexadecimal digits occupy one byte of storage when converted to internal form. By default,

(1) Hexadecimal strings are right-justified.
(2) Hexadecimal strings that are shorter than requested are padded on the left with extra 0's.
(3) Hexadecimal strings that are longer than requested are truncated on the left.
(4) Hexadecimal strings may have an odd or an even number of digits.

Optionally:

(1) By setting the JUSTIFY option to LEFT, hexadecimal strings will be left-justified. Left-justification implies that padding and truncation occur on the right.
(2) By setting the SHORT option to ON, hexadecimal strings that are too short will be treated as errors.
(3) By setting the LONG option to ON, hexadecimal strings that are too long will be treated as errors.
(4) By setting the EVEN option to ON, hexadecimal strings that have an odd number of digits will be treated as errors.

Octal Strings

An octal string is a sequence from the set of characters 0 through 7. One octal digit occupies three bits of storage when converted to internal form. By default:

(1) Octal strings are right-justified to the bit.
(2) Octal strings that are shorter than requested are padded on the left with extra 0's.
(3) Octal strings that are longer than requested are truncated on the left.

Optionally:

(1) By setting the JUSTIFY option to LEFT, octal strings will be left-justified. Left-justification implies that padding and truncation occur on the right.
(2) By setting the SHORT option to ON, octal strings that are too short will be treated as errors. An octal string is considered to be too short when it is more than two bits shorter than the user-supplied string buffer.
(3) By setting the LONG option to ON, octal strings that are too long will be treated as errors.

Binary Strings

A binary string is a sequence of 0's and 1's. A string of eight binary digits occupies one byte of storage when converted to internal form. By default:

(1) Binary strings are right-justified to the bit.
(2) Binary strings that are shorter than requested are padded on the left with extra 0's.
(3) Binary strings that are longer than requested are truncated on the left.

Optionally:

(1) By setting the JUSTIFY option to LEFT, binary strings will be left-justified. Left-justification implies that padding and truncation occur on the right.
(2) By setting the SHORT option to ON, binary strings that are too short will be treated as errors.
(3) By setting the LONG option to ON, binary strings that are too long will be treated as errors.

October 1983

CALLING SUBROUTINES FROM FORTRAN

FORTRAN programs may easily call S-type system subroutines. R-type (register called) subroutines may also be called with a little more difficulty by using the RCALL and ADROF system subroutines.

Each system subroutine called from a FORTRAN program must be called either as a standard subroutine or as a function. If the subroutine uses only the parameter list to accept and return values, it is called as a subroutine using the CALL statement

```
CALL subr(p1,p2,...,pn)
```

If the subroutine returns a value in general register 0 or floating-point register 0, it is called as a function:

```
value = subr(p1,p2,...,pn)
```

where "value" is the value returned. It must be declared to be the appropriate data type (e.g., INTEGER for general register 0 or REAL*4 or REAL*8 for floating-point register 0).

The equivalences of data types for FORTRAN are given below:

Data Type	FORTRAN Declaration
Fullword integer	INTEGER*4 (or INTEGER)
Halfword integer	INTEGER*2
One-Byte integer	LOGICAL*1
8-Byte integer	INTEGER array of two elements
Fullword real	REAL*4 (or REAL)
Doubleword real	REAL*8
Fullword logical	LOGICAL*4 (or LOGICAL) (0 is FALSE, 1 is TRUE)
One-byte logical	LOGICAL*1 (0 is FALSE, 1 is TRUE)
Character string[1]	LOGICAL*1 or REAL*4 array CHARACTER*n (FORTRAN 77 only) ("n" is the length of the string)

Array	Array of appropriate data type
Region	COMMON region of appropriate data types
Variable type	EQUIVALENCEd data types

[1]Only FORTRAN 77 supports a true character data type. FORTRAN 66 (e.g, *FTN-compiled programs) must use either the LOGICAL*1 or REAL*4 data type.

The following programs illustrate FORTRAN calls to system subroutines.

```
CALL MTS
RETURN
END
```

The above example calls the MTS subroutine which requires no parameters.

```
CHARACTER*255 STRING
INTEGER*2 LENGTH
STRING = '$Display Timespelledout'
LENGTH = 23
CALL CMD(STRING,LENGTH)
RETURN
END
```

The above example coded in FORTRAN 77 calls the CMD subroutine to execute a $DISPLAY command. The subroutine requires two parameters, the first being a character string giving the command string and the second being a halfword integer command length (CMD also allows a fullword integer to be used). If this example were used in a FORTRAN 66 program, it could be coded as follows:

```
REAL*4 STRING(6)
INTEGER*2 LENGTH/23/
DATA STRING/'$Dis','play',' Tim','espe','lled','out '/
CALL CMD(STRING,LENGTH)
RETURN
END
```

The second example illustrates the advantage of using FORTRAN 77 when calling subroutines that require character data types. Normally, it is difficult in FORTRAN 66 to process REAL*4 or LOGICAL*1 arrays containing character data without resorting to the use of special character manipulation routines.

```
CHARACTER*4 USERID
INTEGER ITEMNO/2/
CALL GUINFO(ITEMNO,USERID)
WRITE (6,100) USERID
RETURN
```

```
100   FORMAT(' User ID = ',A4)
      END
```

The above example calls the GUINFO subroutine to obtain the current user ID. The subroutine also requires two parameters, the first of which is a fullword integer and the second a character string. In FORTRAN 66, USERID would be declared as REAL*4.

```
      CHARACTER*18 FNAME
      INTEGER CHKFIL,ACCESS
      FNAME = 'WABC:DATA '
      ACCESS = CHKFIL(FNAME)
      WRITE (6,100) ACCESS
      RETURN
100   FORMAT(' Access = ',I2)
      END
```

The above example calls the CHKFIL subroutine to determine the program's access to the file WABC:DATA. Since the access is returned in general register 0, the subroutine must be called as INTEGER function.

```
      IMPLICIT INTEGER(A-Z)
      COMMON /CINFO/ CIAL,CIRL,CIONID,CIVOL,CIUC,CILRD,CICD
      COMMON /CINFO/ CIFO,CIDT,CIFLG,CILCD,CIPKEY,CILCCT
      COMMON /CINFO/ CILNCD,CINCT,CICDT,CILRDT
      CHARACTER*4 CIONID
      CHARACTER*8 CIVOL,UNIT
      CHARACTER*80 ERRMSG
      DIMENSION CILCCT(2),CILNCT(2),CICDT(2),CILRDT(2)
      DIMENSION RTN(6)
      CHARACTER*20 RTNCHR
      EQUIVALENCE (RTN(1),RTNCHR)
      DATA FLAG/Z00000002/
      UNIT = 'SCARDS  '
      RTN(6) = 0
      CIAL = 25
      CALL GFINFO(UNIT,RTNCHR,FLAG,CIAL,0,0,ERCODE,ERRMSG,*20,30)
      WRITE (6,100) RTNCHR,CIONID
      RETURN
20    WRITE (6,101) ERCODE,ERRMSG
      RETURN
30    WRITE (6,102)
      RETURN
100   FORMAT(A20,' Owner = ',A4)
101   FORMAT(I2,2X,A80)
102   FORMAT(' Error return from GFINFO subroutine')
      END
```

The above FORTRAN 77 example calls the GFINFO subroutine to obtain catalog information about the file attached to the logical I/O unit SCARDS. The common block CINFO is passed to the subroutine; upon return, the subroutine will insert the catalog information into this region. By using a common block, a packed region of varying data types

can be defined. The EQUIVALENCE statement is used to equate the integer array RTN with the character string RTNCHR so that RTN may be passed as an integer and returned as a character string.

Return codes from system subroutines are processed by appending statement numbers to the end of the parameter list on the subroutine call. For FORTRAN 66, this takes the form

```
CALL subr(p1,p2,...,pn,&sn1,&sn2,...)
```

and for FORTRAN 77, this takes the form

```
CALL subr(p1,p2,...,pn,*sn1,*sn2,...)
```

Each "sn" specifies a statement number to branch to if the corresponding return code is returned by the system subroutine. For example, the last two parameters in the GFINFO example above, *20 and *30, indicate branches to be made if a nonzero return code occurs, e.g., a branch is made to statement 20 if the return code is 4, and a branch is made to statement 30 if the return code is 8. These are equivalent to the FORTRAN statements RETURN 1 and RETURN 2 (in general, a return code of 4*n is equivalent to RETURN n in FORTRAN). In FORTRAN 66, these branches would be coded as &20 and &30. Note that if a return code occurs that is higher than what is accommodated for in the parameter list, then the next statement after the subroutine call is executed (the same as if the return code were zero). For example, if a return code of 8 occurred and GFINFO were called with the statement

```
CALL GFINFO(UNIT,RTNCHR,FLAG,CIAL,0,0,ERCODE,ERRMSG,*20)
```

then no special branch would be made (*20 does not mean branch to statement 20 for return codes of 4 and higher).

In FORTRAN, there are no special problems connected with calling subroutines that have a variable number of parameters. In the example below, the COMMND subroutine is called once with three parameters and again with five parameters.

```
          IMPLICIT INTEGER(A-Z)
          CHARACTER*255 CMDTXT
          INTEGER SWS/0/
          CMDTXT = '$DISPLAY TIMESPELLEDOUT'
          LENGTH = 23
          CALL COMMND(CMDTXT,LENGTH,SWS,*20,*20,*20)
          CMDTXT = '$DISPLAY TIMEMISSPELLEDOUT'
          LENGTH = 26
          CALL COMMND(CMDTXT,LENGTH,SWS,SUMMRY,ERCODE,*20,*20,*20)
          IF (SUMMRY.GT.0) THEN
            WRITE (6,100) ERCODE
            RETURN
          END IF
          RETURN
    20    WRITE (6,101)
```

```
      RETURN
100   FORMAT(' Command Error Code = ',I3)
101   FORMAT(' Error return from COMMND subroutine')
      END
```

VS FORTRAN (Version 2, Release 2.0 or later) allows external names up to 7 characters. Thus, the above example could be called using the COMMAND entry point instead of the COMMND entry point.

R-Type Subroutines

R-type subroutines can be called from FORTRAN by using the RCALL and ADROF subroutines. The RCALL subroutine sets up a call to an R-type subroutine by inserting the parameters into the proper registers for the call to the system subroutine. The ADROF subroutine is used to obtain the address of a variable as required both for the RCALL subroutine and for other system subroutines such as GETFD.

The call to the RCALL subroutine is made in the following manner:

```
CALL RCALL(subr,r1,p1,...,r2,p2,...)
```

where "r1" is the number of registers to be set up on the call to "subr" and "p1,..." are the values to be inserted into the registers beginning with general register 0; "r2" is the number of registers to contain return values from "subr" and "p2,..." are the variables that will contain the returned values starting with general register 0.

The call to the ADROF subroutine (a function call) is made in the following manner:

```
value = ADROF(par)
```

The value returned by ADROF is the address of the value specified by "par", e.g.,

```
NAMPTR = ADROF('DATA ')
```

returns in NAMPTR the address of the character string 'DATA '. ADROF may also be used to return the address of a vector or a region of storage such as a parameter list (this is illustrated in the second example below).

```
      IMPLICIT INTEGER(A-Z)
      EXTERNAL GETFD
      DATA FIRST/1000/,LAST/1000000000/,BEG/1000/,INC/1000/
      CALL RCALL(GETFD,2,0,ADROF('DATA1 '),1,FDUB)
      CALL RENUMB(FDUB,FIRST,LAST,BEG,INC,*20,*40,*30,*40,*40,*40)
      WRITE (6,100)
      RETURN
```

```
     20    WRITE (6,101)
           RETURN
     30    WRITE (6,102)
           RETURN
     40    WRITE (6,103)
           RETURN
     100   FORMAT(' File successfully renumbered')
     101   FORMAT(' File does not exist')
     102   FORMAT(' Renumber access not allowed')
     103   FORMAT(' Error return from RENUMB subroutine')
           END
```

In the above example, the GETFD subroutine is called to obtain a FDUB-pointer for the file DATA1; the FDUB-pointer is then passed on to the RENUMB subroutine to renumber the file. The GETFD subroutine requires that general register 1 contain the address of the name of the subroutine as returned by the ADROF subroutine. The register count is 2, since RCALL initializes registers beginning with general register 0 (in this case, register 0 is called with a dummy argument of zero). Upon return, GETFD returns the FDUB-pointer in register 0. Hence, the register count is 1 and the FDUB-pointer is inserted in the variable FDUB.

```
           IMPLICIT INTEGER(A-Z)
           EXTERNAL CHKACC
           CHARACTER*18 FNAME
           CHARACTER*26 TRIPLE
           DATA MASK/Z00000006/
           DIMENSION PAR(2)
           FNAME = 'DATA1 '
           TRIPLE = 'WABCWXYZ*EXEC '
           PAR(1) = ADROF(FNAME)
           PAR(2) = ADROF(TRIPLE)
     1     CALL RCALL(CHKACC,2,0,ADROF(PAR),1,ACCESS,*20,*30,*30)
     2     IF (AND(ACCESS,MASK).NE.MASK) THEN
             WRITE (6,100)
             RETURN
           END IF
           WRITE (6,101)
           RETURN
     20    WRITE (6,102)
           RETURN
     30    WRITE (6,103)
           RETURN
     100   FORMAT(' Write access not allowed')
     101   FORMAT(' Write access allowed')
     102   FORMAT(' File does not exist')
     103   FORMAT(' Error return from CHKACC subroutine')
           END
```

The above example illustrates the use of the RCALL subroutine to obtain a return code from a subroutine that would normally be called as a function. When called as a function, i.e.,

```
ACCESS = CHKACC(FNAME,TRIPLE)
```

the CHKACC subroutine returns the access in register 0 and inserts in the variable ACCESS. When called as a CALLed subroutine, i.e.,

```
CALL CHKACC(FNAME,TRIPLE,*20,*30,*30)
```

the access in not available, but return codes are available to determine the success of the call. RCALL can be used to get the best of both worlds. However, since CHKACC is not an R-type subroutine but a standard S-type subroutine, general register 1 must contain the address of the parameter list PAR. This can be accomplished by using ADROF(PAR) to obtain this; in this case, the parameter list PAR is declared as an array of two elements. Note also the use of ADROF to generate the two values in the parameter list, the first being the address of FNAME and the second being the address of TRIPLE.

Special Cases

Several system subroutines cannot be called directly by FORTRAN programs, either because they return storage acquired by the subroutine itself (e.g., GDINFO), or because they require nonstandard calls for exit routines (e.g., ATTNTRP or TIMNTRP). However, most of these subroutines have FORTRAN-callable alternatives that perform similar functions. Some of the more common alternative entries (or subroutines) are given in the table below.

System Subroutine	Alternative Entry
ATTNTRP	ATNTRP
GDINFO	GDINF
LINK	LINKF
LOAD	LOADF
REWIND#	REWIND
TIMNTRP	TICALL
UNLOAD	UNLDF
XCTL	XCTLF

The example below illustrates the use of the GDINF alternative entry to the GDINFO subroutine.

```
IMPLICIT INTEGER(A-Z)
CHARACTER*4 DEVTYP
CHARACTER*8 UNIT
INTEGER*2 INLEN,OUTLEN
LOGICAL*1 USE,DEVICE,SWS1,SWS2
COMMON /INFO/ FDUB,DEVTYP,INLEN,OUTLEN,USE,DEVICE,SWS1,SWS2
COMMON /INFO/ MODS,BEGLNR,PRVLNR,ENDLNR,INCLNR,NAMPTR,MSGPTR
UNIT = 'SCARDS  '
CALL GDINF(UNIT,FDUB,*20,*20)
```

```
      WRITE (6,100) DEVTYP
      RETURN
20    WRITE (6,101)
      RETURN
100   FORMAT(' Type = ',A4)
101   FORMAT(' Error return from GDINF subroutine')
      END
```

October 1983

DYNAMIC LOADING IN FORTRAN

The dynamic loader provides the capability for an executing program to load, execute, and unload another program. This is accomplished by the system subroutines LOAD, LINK, XCTL, and UNLOAD which are described in MTS Volume 3, System Subroutine Descriptions, and the section "Virtual Memory Management" in MTS Volume 5, System Services.

Due to the special nature of a FORTRAN program and its I/O environment, these subroutines cannot be directly used. However, the system does provide equivalent subroutines that FORTRAN programs may use to accomplish the same functions. These subroutines are provided in the form of special entry points to the above-mentioned subroutines. These special entry points are LOADF, LINKF, XCTLF, and UNLDF and they form an interface between a FORTRAN program and their counterpart subroutines LOAD, LINK, XCTL, and UNLOAD. In addition, the subroutine STARTF is provided as a means of invoking a dynamically loaded program. These subroutines interact with the FORTRAN program and its I/O environment so that a dynamically loaded FORTRAN program will execute and return properly.

In general, the interaction between a dynamically loaded FORTRAN program and its calling program is determined by the setting of the "merge" bit in the fullword of switches provided by the subroutine that is loading the program. If the merge bit is 1, both programs will use the same I/O environment. This makes the dynamically loaded program execute the same as if it were called in the form of a normal FORTRAN subroutine which was loaded at the same time as the calling program (except that a STOP statement is treated as a RETURN statement). If the "merge" bit is 0, the dynamically loaded program will use an independent version of the FORTRAN I/O environment.

These dynamic loading subroutines should be used whenever one FORTRAN-compiled program dynamically loads another FORTRAN-compiled program. They may be used also by programs written in other programming languages, but they are intended primarily for use by FORTRAN programs.

LINKF

Purpose: To effect the dynamic loading and execution of a program.

Location: Resident System

Calling Sequence:

```
CALL LINKF(input,info,parlist,errexit,output,lsw,gtsp,
          frsp,pnt)
```

Parameters:

input is the location of an input specifier to be used during loading to read loader records. An input specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 8 in "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an input subroutine to be called during loading via a READ subroutine calling sequence to read loader records (i.e., the input subroutine is called with a parameter list identical to the system subroutine READ). In this case, bit 9 in "info" must be 1.

info is the location of an optional information vector. No information is passed if "info" is 0 or if "info" is the location of a fullword integer 0. The format of the information vector is as follows:

(1) a halfword of LINKF control bits defined as follows:

```
bit 0:   1, if "errexit" is specified.
bit 1:   1, if "output" is specified.
bit 2:   1, if "lsw" is specified.
bit 3:   1, if "gtsp" is specified.
bit 4:   1, if "frsp" is specified.
bit 5:   1, if "pnt" is specified.
bit 6:   1, if to suppress search of LIBSRCH/
            *LIBRARY libraries.
bit 7:   0, unused (must be zero).
bit 8:   1, if  "input"  is the location of a
```

logical I/O unit name.
bit 9: 1, if "input" is the location of an input subroutine address.
bit 10: 1, if "output" is the location of a logical I/O unit name.
bit 11: 1, if "output" is the location of an output subroutine address.
bit 12: 1, if the program to be loaded is to be merged with the program previously loaded.
bit 13: 1, to suppress prompting at a terminal.
bit 14: 1, to force allocation of a new loader symbol table.
bit 15: 0

(2) a halfword count of the number of entries in the following initial ESD list.
(3) a variable-length initial ESD list, each entry of which consists of a fullword-aligned 8-character symbol followed by a fullword value.

parlist is the location of a parameter list to be passed in GR1 to the program being linked to.

errexit (optional) is the location of an error-exit subroutine address to be called if an error occurs while attempting to link to the specified program. If bit 0 of "info" is 0 (the default), the "errexit" parameter is ignored and an error return is made to MTS command mode. The exit routine will be called via a standard S-type calling sequence with two parameters defined as follows:

P1: the location of a fullword integer error code defined as follows:

0: attempt to load a null program.
4: fatal loading error (bad object program).
8: undefined symbols referenced by the loaded program.
12: no available storage index numbers.
16: maximum number of link levels exceeded.

P2: the location of a fullword containing the loader status word.

If the exit routine returns, LINKF will return to MTS without releasing program storage (i.e., as if the error exit had not been taken).

output (optional) is the location of an output specifier to be used during loading to produce loader output (error messages, map, etc.). If bit 1 of "info" is 0 (the default), the "output" parameter is ignored and all loader output is written on the MAP=FDname specified on the initial $RUN command. An output specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 10 of "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an output subroutine to be called during loading via an SPRINT subroutine calling sequence to write loader output (i.e., the output subroutine is called with a parameter list identical to the system subroutine SPRINT). In this case, bit 11 of "info" must be 1.

lsw (optional) is the location of a fullword of loader control bits. If bit 2 of "info" is 0 (the default), the "lsw" parameter is ignored and the global MTS settings are used. The loader control bits are defined as follows:

bits 0-23: 0
bit 24: 1, to suppress the pseudoregister map.
bit 25: 1, to suppress the predefined symbol map.
bit 26: 1, to print undefined symbols.
bit 27: 1, to print references to undefined symbols.
bit 28: 1, to print references to all external symbols.
bit 29: 1, to print dotted lines around the loader map.
bit 30: 1, to print a map.
bit 31: 1, to print nonfatal error messages.

gtsp (optional) is the location of a storage allocation subroutine to be called during loading via a GETSPACE calling sequence to allocate program storage. If bit 3 of "info" is zero (the default), GETSPACE is used.

frsp (optional) is the location of a storage deallocation subroutine to be called during loading via a FREESPAC calling sequence to release loader work space. If bit 4 of "info" is 0 (the default), FREESPAC is used.

pnt (optional) is the location of a direct access subroutine to be called during loading via a POINT calling sequence while processing libraries in sequential files. If bit 5 of "info" is 0 (the default), POINT is used.

Values Returned:

None.

Description: LINKF provides a method for dynamically loading and executing a program. LINKF provides this facility as follows:

(1) The loader is called to dynamically load the specified program using "input", "info", "output", "lsw", "gtsp", "frsp", and "pnt" if specified.
(2) The dynamically loaded program is called with the address of "parlist" in GR1.
(3) If the dynamically loaded program, returns to LINKF, it is unloaded.
(4) LINKF returns to the calling program preserving the return registers of the dynamically executed program.

Note that LINKF accepts a variable-length parameter list of three to eight arguments. For most applications, only the first three are required.

LINKF is required to provide the dynamically loaded program with a FORTRAN I/O environment consistent with the "merge" bit specified in "info". If the merge bit is 1, the dynamically loaded program will have the same I/O environment as the calling program. If the merge bit is 0, the dynamically loaded program will have a separate, reinitialized I/O environment. Both FORTRAN main programs and subroutines can be dynamically loaded using LINKF. However, the effect of executing a STOP statement from a dynamically loaded subroutine will depend on the setting of the merge bit. If the merge bit is 1, a return is made to the calling program; if the merge bit is 0, a return is made to MTS.

Because the rate structure for use of MTS includes a charge for allocated virtual memory integrated over CPU time, the cost of running a large software package in MTS can often be reduced by dynamically loading and executing seldom-used subroutines via a call to LINKF. Such savings in the storage integral must be weighed against the additional CPU time required to open a second file, reinvoke the loader, and rescan the required libraries.

The user also should see the sections "The Dynamic Loader" and "Virtual Memory Management" in MTS Volume 5, System Services. In particular, these sections describe the use of initial ESD lists, merging with previously loaded programs, and the relationship between LINKF, LOADF, and XCTLF storage management.

Example:

```
      INTEGER*2 PAR(4)
      INTEGER*4 ADROF
      DATA PAR/6,'*T','P1','* '/
      CALL LINKF('*LABELSNIFF ',0,ADROF(PAR))
      END
```

The above FORTRAN program is equivalent to issuing the MTS command "$RUN *LABELSNIFF PAR=*TP1*".

XCTLF

Purpose: To effect the dynamic loading and execution of a program.

Location: Resident System

Calling Sequence:

```
CALL XCTLF(input,info,parlist,errexit,output,lsw,gtsp,
          frsp,pnt)
```

Parameters:

input is the location of an input specifier to be used during loading to read loader records. An input specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 8 in "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an input subroutine to be called during loading via a READ subroutine calling sequence to read loader records (i.e., the input subroutine is called with a parameter list identical to the system subroutine READ). In this case, bit 9 in "info" must be 1.

info is the location of an optional information vector. No information is passed if "info" is 0 or if "info" is the location of a fullword integer 0. The format of the information vector is as follows:

(1) a halfword of XCTLF control bits defined as follows:

```
bit 0:  1, if "errexit" parameter is
           specified.
bit 1:  1, if "output" is specified.
bit 2:  1, if "lsw" is specified.
bit 3:  1, if "gtsp" is specified.
bit 4:  1, if "frsp" is specified.
bit 5:  1, if "pnt" is specified.
bit 6:  1, if to suppress search of LIBSRCH/
           *LIBRARY libraries.
bit 7:  1, to request XCTLF to restore the
```

registers of the previous link level before transferring control to the specified program.
0, if the caller has restored them.
bit 8: 1, if "input" is the location of a logical I/O unit name.
bit 9: 1, if "input" is the location of an input subroutine address.
bit 10: 1, if "output" is the location of a logical I/O unit name.
bit 11: 1, if "output" is the location of an output subroutine address.
bit 12: 1, if the program to be loaded is to be merged with the program previously loaded.
bit 13: 1, to suppress prompting at a terminal.
bit 14: 1, to force allocation of a new loader symbol table.
bit 15: 0

(2) a halfword count of the number of entries in the following initial ESD list.
(3) a variable-length initial ESD list, each entry of which consists of a fullword-aligned 8-character symbol followed by a fullword value.

parlist is the location of a parameter list to be passed in GR1 to the program being transferred to.

errexit (optional) is the location of an error-exit subroutine address to be called if an error occurs while attempting to transfer to the specified program. If bit 0 of "info" is 0 (the default), the "errexit" parameter is ignored and an error return is made to MTS command mode. The exit routine will be called via a standard S-type calling sequence with two parameters defined as follows:

P1: the location of a fullword integer error code defined as follows:

0: attempt to load a null program.
4: fatal loading error (bad object program).
8: undefined symbols referenced by the loaded program.

P2: the location of a fullword containing the loader status word.

If the exit routine returns, XCTLF will return to MTS without releasing program storage (i.e., as if the error exit had not been taken).

output (optional) is the location of an output specifier to be used during loading to produce loader output (error messages, map, etc.). If bit 1 of "info" is 0 (the default), the "output" parameter is ignored and all loader output is written on the MAP=FDname specified on the initial $RUN command. An output specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 10 of "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an output subroutine to be called during loading via an SPRINT subroutine calling sequence to write loader output (i.e., the output subroutine is called with a parameter list identical to the system subroutine SPRINT). In this case, bit 11 of "info" must be 1.

lsw (optional) is the location of a fullword of loader control bits. If bit 2 of "info" is 0 (the default), the "lsw" parameter is ignored and the global MTS settings are used. The loader control bits are defined as follows:

bits 0-23: 0
bit 24: 1, to suppress the pseudoregister map.
bit 25: 1, to suppress the predefined symbol map.
bit 26: 1, to print undefined symbols.
bit 27: 1, to print references to undefined symbols.
bit 28: 1, to print references to all external symbols.
bit 29: 1, to print dotted lines around the loader map.
bit 30: 1, to print a map.
bit 31: 1, to print nonfatal error messages.

gtsp (optional) is the location of a storage allocation subroutine to be called during loading via a GETSPACE calling sequence to allocate program storage. If bit 3 of "info" is zero (the default), GETSPACE is used.

frsp (optional) is the location of a storage deallocation subroutine to be called during loading via a FREESPAC calling sequence to release loader work space. If bit 4 of "info" is 0 (the default), FREESPAC is used.

pnt (optional) is the location of a direct access subroutine to be called during loading via a POINT calling sequence while processing libraries in sequential files. If bit 5 of "info" is 0 (the default), POINT is used.

Values Returned:

None.

Description: XCTLF provides a method for dynamically loading and executing programs in an overlay fashion. XCTLF provides this facility as follows:

(1) XCTLF makes a copy of all its parameter values and releases all storage associated with the current link level.
(2) The loader is called to dynamically load the specified program using "input", "info", "output", "lsw", "gtsp", "frsp", and "pnt" if specified.
(3) The dynamically loaded program is called with the address of "parlist" in GR1.
(4) If the dynamically loaded program returns to XCTLF, it is unloaded.
(5) XCTLF returns to the program which initiated the current link level, preserving the return registers of the dynamically executed program.

Note that XCTLF accepts a variable-length parameter list of three to eight arguments. For most applications, only the first three are required. These parameters passed to XCTLF may be part of the current link level to be released, since XCTLF makes copies of them. However, the parameter list and parameters passed to the program XCTLed to, as well as the optional subroutines specified by "input", "output" "errexit", "gtsp", "frsp", and "pnt" may not be part of the current link level since it is released before the program transferred to, is loaded and executed.

Note that by default it is the user's responsibility to restore the registers of the previous link level before calling XCTLF. Since in general this is possible only at the assembly language level, calls to XCTL should have bit 7 in "info" set to 1, regardless of its setting in the "info" parameter.

XCTLF is required to provide the dynamically loaded program with a FORTRAN I/O environment consistent with the "merge" bit specified in "info". If the merge bit is 1, the dynamically loaded program will have the same I/O environment as the calling program. If the merge bit is 0, the dynamically loaded program will have a separate, reinitialized I/O environment. Both FORTRAN main programs and subroutines can be dynamically loaded using XCTLF. However, the effect of executing a STOP statement from a dynamically loaded subroutine will depend on the setting of the merge bit. If the merge bit is 1, a return is made to the program which linked to the calling program; if the merge bit is 0, a return is made to MTS.

Because the rate structure for use of MTS includes a charge for allocated virtual memory integrated over CPU time, the cost of running a large software package in MTS can often be reduced by dynamically loading and executing sequential phases in an overlay fashion via calls to XCTLF. Such savings in the storage integral must be weighed against the additional CPU time required to open a second file, reinvoke the loader, and rescan the required libraries.

The user also should see the sections "The Dynamic Loader" and "Virtual Memory Management" in MTS Volume 5, System Services. In particular, they describe the use of initial ESD lists, merging with previously loaded programs, and the relationship between LINKF, LOADF, and XCTLF storage management.

LOADF

Purpose: To effect the dynamic loading of a program.

Location: Resident System

Calling Sequence:

```
indx = LOADF(input,info,switch,rtnlist,output,lsw,gtsp,
            frsp,pnt)
```

Parameters:

input is the location of an input specifier to be used during loading to read loader records. An input specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 8 in "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an input subroutine to be called during loading via a READ subroutine calling sequence to read loader records (i.e., the input subroutine is called with a parameter list identical to the system subroutine READ). In this case, bit 9 in "info" must be 1.

info is the location of an optional information vector. No information is passed if "info" is 0 or if "info" is the location of a fullword integer 0. The format of the information vector is as follows:

(1) a halfword of LOADF control bits defined as follows:

```
bit 0:   1, if "rtnlist" is to be ignored.
bit 1:   1, if "output" is specified.
bit 2:   1, if "lsw" is specified.
bit 3:   1, if "gtsp" is specified.
bit 4:   1, if "frsp" is specified.
bit 5:   1, if "pnt" is specified.
bit 6:   1, if to suppress search of LIBSRCH/
            *LIBRARY libraries.
bit 7:   0, unused (must be zero).
bit 8:   1, if  "input"  is the location of a
```

logical I/O unit name.
bit 9: 1, if "input" is the location of an input subroutine address.
bit 10: 1, if "output" is the location of a logical I/O unit name.
bit 11: 1, if "output" is the location of an output subroutine address.
bit 12: 1, if the program to be loaded is to be merged with the program previously loaded.
bit 13: 1, to suppress prompting at a terminal.
bit 14: 1, to force allocation of a new loader symbol table.
bit 15: 0

(2) a halfword count of the number of entries in the following initial ESD list.
(3) a variable-length initial ESD list, each entry of which consists of a fullword-aligned 8-character symbol followed by a fullword value.

switch is the location of a fullword of LOADF control bits defined as follows:

bits 0-7: the storage index number to be used if bit 29 or 30 is 1; else, optionally, the number of the segment into which the program is to be loaded.
bit 8: 1, if "rtnlist" is to be ignored.
bit 9: 1, if "output" is specified.
bit 10: 1, if "lsw" is specified.
bit 11: 1, if "gtsp" is specified.
bit 12: 1, if "frsp" is specified.
bit 13: 1, if "pnt" is specified.
bits 14-19: 0
bit 20: 1, if "input" is the location of a logical I/O unit name.
bit 21: 1, if "input" is the location of an input subroutine address.
bit 22: 1, if "output" is the location of a logical I/O unit name.
bit 23: 1, if "output" is the location of an output subroutine address.
bit 24: 0
bit 25: 1, if the program to be loaded is to be merged with those previously loaded.
bit 26: 1, to return if a loading error occurs.
0, to call MTS if a loading error occurs.
bit 27: 1, to suppress prompting at a terminal.
bit 28: 1, to force allocation of a new loader symbol table.

bit 29: 1, to load using the storage index number specified in bits 0-7.
bit 30: 1, load into system storage (bits 0-7 contain the storage index number to be used). This bit is only valid for systems programs.
bit 31: 0, load at the highest link level;
1, load at the current link level.

rtnlist is either 0 or the address of an area into which the loader will place an ESD list of all the symbols in the loader symbol table.

output (optional) is the location of an output specifier to be used during loading to produce loader output (error messages, map, etc.). If bit 1 of "info" is 0 (the default), the "output" parameter is ignored and all loader output is written on the MAP=FDname specified on the initial $RUN command. An output specifier may be one of the following:

(1) an FDname terminated by a blank.
(2) a FDUB-pointer (as returned by GETFD).
(3) an 8-character logical I/O unit name, left-justified with trailing blanks. In this case, bit 10 of "info" must be 1.
(4) a fullword integer logical I/O unit number (0-99).
(5) the address of an output subroutine to be called during loading via an SPRINT subroutine calling sequence to write loader output (i.e., the output subroutine is called with a parameter list identical to the system subroutine SPRINT). In this case, bit 11 of "info" must be 1.

lsw (optional) is the location of a fullword of loader control bits. If bit 2 of "info" is 0 (the default), the "lsw" parameter is ignored and the global MTS settings are used. The loader control bits are defined as follows:

bits 0-23: 0
bit 24: 1, to suppress the pseudoregister map.
bit 25: 1, to suppress the predefined symbol map.
bit 26: 1, to print undefined symbols.
bit 27: 1, to print references to undefined symbols.
bit 28: 1, to print references to all external symbols.
bit 29: 1, to print dotted lines around the loader map.

bit 30: 1, to print a map.
bit 31: 1, to print nonfatal error messages.

gtsp (optional) is the location of a storage allocation subroutine to be called during loading via a GETSPACE calling sequence to allocate program storage. If bit 3 of "info" is zero (the default), GETSPACE is used.

frsp (optional) is the location of a storage deallocation subroutine to be called during loading via a FREESPAC calling sequence to release loader work space. If bit 4 of "info" is 0 (the default), FREESPAC is used.

pnt (optional) is the location of a direct access subroutine to be called during loading via a POINT calling sequence while processing libraries in sequential files. If bit 5 of "info" is 0 (the default), POINT is used.

Values Returned:

If loading was successful, a positive INTEGER*4 storage index number is returned as the value of LOADF. This number is used to uniquely identify the dynamically loaded program on subsequent calls to STARTF and UNLDF.

If a loading error occurred, a negative INTEGER*4 error code is returned as the value of LOADF, and is defined as follows:

-1: attempt to load a null program.
-2: fatal loading error (bad object program).
-3: undefined symbols referenced by the loaded program.
-4: no available storage index numbers.

Description: LOADF provides a method for dynamically loading a program. LOADF provides this facility as follows:

(1) The loader is called to dynamically load the specified program using "input", "info", "output", "lsw", "gtsp", "frsp", and "pnt" if specified.
(2) LOADF returns to the calling program with the return values described above.

Note that LOADF accepts a variable-length parameter list of 4 to 8 arguments. For most applications, only the first 4 are required. Both "info" and "switches" contain LOADF control bits, some of which are duplicates. In

these cases, LOADF produces a single control bit by "OR"ing the two together.

LOADF is required to provide the dynamically loaded program with a FORTRAN I/O environment consistent with the "merge" bit specified in "info". If the "merge" bit is 1, the dynamically loaded program will have the same I/O environment as the calling program. If the "merge" bit is 0, the dynamically loaded program will have a separate, reinitialized I/O environment. Both FORTRAN main programs and subroutines can be dynamically loaded using LOADF. However, the effect of executing a STOP statement from a dynamically loaded subroutine will depend on the setting of the "merge" bit. If the "merge" bit is 1, a return is made to the calling program; if the "merge" bit is 0, a return is made to MTS. LOADF returns an INTEGER*4 storage index number used to uniquely identify the dynamically loaded program on subsequent calls to STARTF and UNLDF.

Because the rate structure for use of MTS includes a charge for allocated virtual memory integrated over CPU time, the cost of running a large software package in MTS can often be reduced by dynamically loading and executing seldom-used subroutines via a call to LOADF. Such savings in the storage integral must be weighed against the additional CPU time required to open a second file, reinvoke the loader, and rescan the required libraries.

The user also should see the sections "The Dynamic Loader" and "Virtual Memory Management" in MTS Volume 5, System Services. In particular, they describe the use of initial ESD lists, merging with previously loaded programs, and the relationship between LOADF, LINKF, and XCTLF storage management.

Example:

```
      LOGICAL*1 PAR(8)
      DATA PAR/'H','I',' ','T','H','E','R','E'/
      INTEGER SWITCH/Z00800040/
      INTEGER*2 LPAR(5)/8/
      EQUIVALENCE (LPAR(2),PAR)

      ID = LOADF('FORTOBJ ',0,SWITCH,0)
      CALL STARTF(ID,LPAR)
      CALL UNLDF(0,ID,0)
```

The above FORTRAN program dynamically loads the program in the file FORTOBJ at the highest link level with the "merge" bit set to 1. Subsequently, the loaded program is executed via a call to STARTF and unloaded via a call to UNLDF.

October 1983

STARTF

Purpose: To execute a program dynamically loaded by the subroutine LOADF.

Location: Resident System

Calling Sequence:

CALL STARTF(id,par1,par2,...)

Parameters:

id is the location of the fullword integer storage index number of the program that was dynamically loaded by LOADF (the value returned by LOADF), or is the location of an 8-character entry point name, left-justified with trailing blanks.

par1,par2,... (optional) are the parameters to be passed to the program being executed. There may be any number of parameters passed, including none.

Values Returned:

None.

Description: STARTF is used to execute a program loaded by the subroutine LOADF. STARTF should be used whenever the calling program and the program being called are FORTRAN programs or programs which use the FORTRAN I/O library. This is necessary in order to provide the proper I/O environment for both the called program and the calling program on return. In providing this, the I/O library environment is established in accordance with the "merge" bit. If the merge bit is 1, then both the calling and called programs use the same I/O library environment; if the merge bit is 0, then the calling and called programs each use a separate copy of the I/O library environment, thus performing relatively independent I/O operations.

If "id" is a storage index number, the dynamically loaded program at that storage index number is invoked at the entry point determined by the loader. If "id" is a symbol, and if the MTS global SYMTAB option is ON, the dynamically loaded program is invoked at the location associated with that symbol in the loader symbol table.

Example:

```
INTEGER*4 PAR1/'ARG1'/,PAR2/'ARG2'/
INTEGER*4 INFO/Z80000000/,SWITCH/Z00000040/
ID = LOADF('FORTOBJ ',INFO,SWITCH,0)
CALL STARTF(ID,PAR1,PAR2)
CALL UNLDF('FORTOBJ ',0,0)
```

This example loads the program in the file FORTOBJ and executes it. The merge bit is set to 1 so that both programs use the same I/O environment.

October 1983

UNLDF

Purpose: To unload a program that was dynamically loaded by the subroutine LOADF.

Location: Resident System

Calling Sequence:

CALL UNLDF(name,id,switch,&rc4)

Parameters:

name is either the location of the "name" (specified by "switch") or zero.

id is either the location of a fullword storage index number or zero. This parameter is referenced only if "name" is zero.

switch is the location of a fullword switch whose value may be one of the following:

0 "name" is the FDname from which program is to be loaded.
1 "name" is an external symbol, 8 characters in length, left-justified with trailing blanks (the MTS global SYMTAB option must be ON).
2 "name" is a fullword virtual address (the MTS global SYMTAB option must be ON).

&rc4 is the statement label to transfer to if a nonzero return code is given.

Return Codes:

0 Successful return.
4 The subroutine could not find the name in the LOAD table, or "switch" is nonzero and SYMTAB is OFF, or the external symbol or virtual memory address could not be found in the loader tables.

Description: The UNLDF subroutine may be used to unload programs dynamically loaded by the LOADF subroutine. Each time the LOADF subroutine is called, a new storage index number is assigned for use with storage acquired in order to load the program specified for that LOADF call. In order to unload the program, either the storage index number or the name of the FDname loaded from may be given. In addition, if the MTS global SYMTAB option is ON, the name of an external symbol or a virtual address in the program loaded may be specified. In any case, the entire program loaded on that call to LOADF is unloaded.

October 1983

October 1983

ARRAY MANAGEMENT SUBROUTINES

This subroutine package permits FORTRAN users to create, extend, and erase 1- and 2-dimensional arrays at execution time. The package resides in *LIBRARY.

Any program or subroutine which references an array created by AMS must include an appropriate subset of the following statements:

```
LOGICAL*1 $L1(1)
LOGICAL*4 $L4(1)
INTEGER*2 $I2(1)
INTEGER*4 $I4(1)
REAL*4 $R4(1)
REAL*8 $R8(1)
COMPLEX*8 $C8(1)
EQUIVALENCE ($L1(1),$L4(1),$I2(1),$I4(1),$R4(1),$R8(1),$C8(1))
COMMON /$/ $I4
```

The above statements establish a set of names called base names, all of which reference the same address in memory.

An ordinary FORTRAN array element is addressed in the form:

```
array name(index)
```

An AMS array element is addressed in the form:

```
base name(array name + index)
```

where the base name should match the FORTRAN type of the array. For example, an INTEGER*4 FORTRAN array named ALPHA might be referenced as ALPHA(I). An AMS array of the same name and type should be referenced as $I4(ALPHA+I). If the array type is REAL*8, it should be referenced as $R8(ALPHA+I) and so on for the other array types.

Other base names may be used instead, but the above names are recommended as they serve to remind the user of the type of array being referenced. Starting the base names with a dollar sign ($) serves to make references to these arrays conspicuous in the program listing. Base names need not be defined for any array types not used by the program, except that an INTEGER*4 base must be named and passed in COMMON /$/ even if the user creates no INTEGER*4 arrays.

If the above declarations are properly made, then an AMS array may be passed to a subroutine merely by passing its array name, either as an argument or in COMMON.

The user-callable subroutines in AMS are:

Name	Purpose
ARINIT	to initialize AMS
ARRAY	to create a 1-dimensional array
ARRAY2	to create a 2-dimensional array
EXTEND	to extend a 1-dimensional array
XTEND2	to extend a 2-dimensional array
ERASE	to erase a single array
ERASAL	to erase all arrays

All arguments passed to and returned by these routines must be INTEGER*4 values.

AMS calls in turn the MTS subroutines GETSPACE, FREESPAC, IMVC, and ADROF.

Note to users who are doing dynamic program loading via LINKF, LOADF, and XCTLF: the storage obtained by AMS will be associated with the highest-level program and will not be released until execution is terminated. To release unwanted arrays, call ERASE or ERASAL.

Warning: The subroutines will not work properly if called from *WATFIV or *IF.

October 1983

ARINIT

Purpose: Before any arrays are created, the user must make one and only one call to subroutine ARINIT. This routine initializes AMS, mainly by creating an array called the master table, which is used by AMS to keep track of the user's arrays. The user does not have direct access to the master table.

Calling Sequence:

```
CALL ARINIT(noar,minc,&s1,&s2,&s3)
```

Parameters:

- noar — an integer in the range 1 to 37449, which specifies the number of arrays the user expects to create during the job. This is an estimate and not an upper limit.
- minc — a positive integer specifying the number of arrays that the master table should be extended to accommodate in case it overflows. It will be automatically extended by this amount an indefinite number of times, as needed.

Return Codes:

Normal	Initialization successful.
&s1	No space available to create master table.
&s2	Invalid argument passed (i.e., noar not in range or minc not positive).
&s3	ARINIT already has been called successfully.

Example:

```
CALL ARINIT(100,50,&98,&99)
```

The master table is created with enough room to handle 100 arrays. Should more arrays be requested, the master table will be automatically extended to accommodate another 50 arrays. If any time during the run the master table should overflow again, it will be extended to accommodate yet another 50 arrays. Control will pass to statement 98 in the user's program if memory space is not available to create the master table. Control will pass to statement 99 if an invalid argument is passed.

ARRAY, ARRAY2

Purpose: To create a 1-dimensional array, ARRAY should be called. To create a 2-dimensional array, ARRAY2 should be called.

Calling Sequences:

```
CALL ARRAY(n,t,d1,&s1,&s2,&s3,&s4)
CALL ARRAY2(n,t,d1,d2,&s1,&s2,&s3,&s4)
```

Parameters:

- t — length in bytes of an array element (1, 2, 4 or 8).
- d1 — a positive integer specifying the number of elements in the 1st dimension of the array.
- d2 — a positive integer specifying the number of elements in the 2nd dimension of the array.

Note: The number of bytes in the array will be t*d1*d2, and this product must be in the range 1 to 1048576.

Values Returned:

- n — name of array to be created. The integer value returned will be such that when n is used in the array reference "base name(n+i)", the "i"th element of the array will be referenced (base name = $L1, $L4, $I2, $I4, $R4, $R8 or $C8.)

  When creating a 1-dimensional array, argument n may take the form of an undimensioned FORTRAN variable such as N, a FORTRAN array element such as N(J), or an AMS array element such as $I4(N+J). In any case, n must be of type INTEGER*4.

  When creating a 2-dimensional array, argument n may not take the form of an undimensioned variable. It must be the first element of either a FORTRAN or an AMS INTEGER*4 array dimensioned at least d2 in length. This is the user's responsibility.

Return Codes:

- Normal — Array created successfully.
- &s1 — Requested array size out of range.

&s2 No space available for requested array. No new arrays may be created unless some existing arrays are erased.

&s3 Request for extension of master table is greater than 1048576 bytes.

&s4 t is not equal to 1, 2, 4 or 8, or ARINIT was never called.

Examples: The following examples illustrate the creation of 1-dimensional arrays:

```
(1)  CALL ARRAY(N,1,100,&1,&2,&3,&4)

         To reference "i"th element:  $L1(N+I)

(2)  INTEGER*4 N(20)
      ...
     CALL ARRAY(N(J),8,250)

         To reference "i"th element:  $R8(N(J)+I)

(3)  CALL ARRAY(N,4,20)
      ...
     CALL ARRAY($I4(N+J),2,1500)

         To reference "i"th element:  $I2($I4(N+J)+I)
```

Note that by the method of the second and third examples, a series of independent arrays may be created, all referenced by the same name, but by different values of J. This is like having a 2-dimensional array where each column may be of a different type and length and may be created, extended, or erased independently. This is useful if the exact number of arrays required by a program is unknown until determined by execution-time data or calculation.

The following examples illustrate the creation of 2-dimensional arrays:

```
(4)  INTEGER*4 N(20)
      ...
     CALL ARRAY2(N(1),4,200,20)

         To reference element "i,j":  $R4(N(J)+I)

(5)  CALL ARRAY(N,4,20)
      ...
     CALL ARRAY2($I4(N+1),8,3000,20)

         To reference element "i,j":  $R8($I4(N+J)+I)
```

EXTEND, XTEND2

Purpose: To extend a 1-dimensional array, EXTEND should be called. To extend a 2-dimensional array, XTEND2 should be called. This routine allocates new space dimensioned according to the request, moves the contents of the old space to the new space, calculates new name values for the new space, and frees the old space.

Calling Sequences:

```
CALL EXTEND(n,inc1,&s1,&s2,&s3)
CALL XTEND2(n,inc1,inc2,&s1,&s2,&s3)
```

Parameters:

n	name of array to be extended.
inc1	a positive integer or zero specifying the number of array elements to be added to 1st dimension of array.
inc2	a positive integer or zero specifying the number of array elements to be added to 2nd dimension of array.

Note: inc1 and inc2 may not both be zero.

Values Returned:

n	new name value for new space obtained.

Return Codes:

Normal	Array extended successfully.
&s1	Size of extended array is greater than 1048576 bytes.
&s2	No space available for extension of array.
&s3	Invalid argument (i.e., array name not recognized, negative inc1 or inc2, or inc1 and inc2 both zero), or ARINIT was never called.

Examples:

```
CALL EXTEND(ALPHA,500,&9,&10,&11)
CALL EXTEND(BETA,M)
CALL XTEND2($I4(A+1),M,0)
CALL XTEND2($I4(A+1),M,N)
```

Note: When extending a two-dimensional array in the second dimension, the argument n (the array name) must be the first element of an array dimensioned at least d2 in length. If the array containing n is not as long as the new expected value of d2, the array containing n must be

extended before the two-dimensional array to which it refers is extended. For example,

```
CALL ARRAY(N,4,20)
 ...
CALL ARRAY2($I4(N+1),8,3000,20)
 ...
CALL EXTEND(N,30)
CALL XTEND2($I4(N+1),0,30)
```

ERASE

Purpose: This routine may be called to erase an array.

Calling Sequence:

```
CALL ERASE(n,&s1)
```

Parameters:

n	name of array to be erased.

Values Returned:

n	A value of -1 is returned to enable both the user and AMS to check if an array has been erased.

Return Codes:

Normal	Array erased successfully.
&s1	Array name not recognized, or ARINIT was never called.

Examples:

```
CALL ERASE(X)
CALL ERASE(ABC,&99)
CALL ERASE($I4(XYZ+1),&100)
```

ERASAL

Purpose: This routine may be called to erase all arrays. New arrays may subsequently be created without recalling ARINIT. (In fact, ARINIT should never be called more than once in the same run.)

Calling Sequence:

```
CALL ERASAL
```

October 1983

CHARACTER MANIPULATION ROUTINES

This subroutine package provides a character manipulation capability for FORTRAN programs. The package resides in *LIBRARY.

The character manipulation routines have the following entry points: BTD, COMC, DTB, EQUC, FINDC, FINDST, IGC, LCOMC, MOVEC, SETC, TRNC, TRNST.

The subroutines described in this section make use of the character orientation of the IBM System 360/370 and the fact that each character can be referenced in a LOGICAL*1 array in a FORTRAN program. Subroutines are available for searching for characters or character strings, ignoring characters, translating characters or character strings, moving characters, and comparing character strings. All of these subroutines are written in 360-assembler language. It is possible to write FORTRAN equivalents of each, but at the expense of both CPU time and virtual memory space.

Four of the routines, FINDC, FINDST, IGC, and TRNST, return a position in a LOGICAL*1 array as an argument. In order that this position be relative to the start of the array, these routines have a slightly more cumbersome calling sequence than the other routines. This approach was dictated by the fact that routines which return positions relative to the start of a search (which may not be the start of an array) result in many programming errors due to misunderstandings about the positions returned.

Three of the routines, FINDC, IGC, and TRNC, search for characters. In order for the search to be carried out, an initialization step, which may take more CPU time than the search itself, is made. Since the initialization is the same for any given set of characters or character string, these routines allow the user to indicate whether the same characters are to be used again. If the expression indicating the number of characters is set to zero, the same characters given on the last nonzero call will be used. This saves repeating the initialization step. Users should try to take advantage of this in their programs.

While the subroutines were designed with the use of LOGICAL*1 variables in mind, knowledgeable users can, in fact, use them to manipulate characters stored in any type of FORTRAN variable.

These routines typically require a fraction of a millisecond of CPU time. This depends a great deal on the number of characters involved, but timings greater than one-half millisecond are rare. The virtual memory required averages about 250 bytes per routine.

The following terms are used in the subroutine descriptions that follow:

array variable

> The name of a dimensioned variable or element of a dimensioned variable.

INTEGER expression

> Any valid INTEGER constant (e.g., 10), variable name (e.g., I), or arithmetic expression (e.g., I+3, 4*K+12).

LOGICAL*1 character array

> A dimensioned LOGICAL*1 variable containing character information.

October 1983

BTD

Purpose: To convert FORTRAN INTEGER numbers into numeric character strings.

Calling Sequence:

FORTRAN: CALL BTD(integer,to,cnumb,dnumb,fill,&err)

Parameters:

- integer is an INTEGER expression giving the number to be converted.
- to is a LOGICAL*1 array variable indicating the position at which the first character is to be stored.
- cnumb is an INTEGER expression giving the number of characters in the string. cnumb should be ≤ 12 and ≥ 0. If cnumb=0, then the number of characters will be the number of significant digits in integer plus one for the sign if integer is negative. If cnumb>12, the characters will be right-justified in the 12 positions starting with to and a RETURN 1 will be taken.
- dnumb is an INTEGER variable which will be set to the number of significant digits in integer (plus one if the sign is negative).
- fill is a LOGICAL*1 character variable, or a Hollerith literal, giving a character to be used to replace leading zeros in the string.
- err (optional) is the number of a FORTRAN statement to transfer to if cnumb>12.

Comments: After a call to BTD, dnumb>cnumb implies a loss of significant digits in the conversion.

If integer equals zero, then the entire field of cnumb characters, starting with the character specified by to, will consist of fill characters.

Example: The example below converts the integer I into a 7-character string with leading zeros replaced by percent signs (%).

```
LOGICAL*1 CHAR(10)
CALL BTD(I,CHAR(1),7,ND,'%')
```

If I=-84, the 7 characters stored in CHAR(1) to CHAR(7) will be %%%%-84. ND will be set to 3.

COMC

Purpose: To determine whether one character string is less than, equal to, or greater than, another string.

Calling Sequence:

FORTRAN: CALL COMC(numb,string1,string2,differ,&err1, &err2,&err3)

Parameters:

- numb is an INTEGER expression giving the number of characters in each string.
- string1,string2 are the character strings to be compared for equality and may be specified either by an array variable or by a Hollerith literal. Equality is interpreted in the sense of position within the 360 collating sequence.
- differ is an INTEGER variable which is set to the position of the first character in string1 which differs from the corresponding character in string2. If string1 and string2 are identical, differ is set to zero.
- err1 (optional) is the number of a FORTRAN statement to transfer to if string1<string2, i.e., if string1 precedes string2 in the collating sequence.
- err2 (optional) is the number of a FORTRAN statement to transfer to if string1>string2, i.e., if string1 follows string2 in the collating sequence.
- err3 (optional) is the number of a FORTRAN statement to transfer to if numb≤0.

Comments: The first character that differs dictates whether string1 is less than or greater than string2. If this character in string1 appears in the collating sequence before the corresponding character in string2, then string1<string2; otherwise, string1>string2. A normal RETURN is made if string1 is identical to string2. If numb≤0, no comparison is made.

Example: The example below compares the 9 characters starting at A(15) with the character string PAR FIELD and branches to statement number 12 on inequality.

```
LOGICAL*1 A(50)
CALL COMC(9,'PAR FIELD',A(15),IDIF,&12,&12)
```

DTB

Purpose: To convert a string of numeric characters into a FORTRAN INTEGER number.

Calling Sequence:

FORTRAN: CALL DTB(from,integer,cnumb,dnumb,fill,&err)

Parameters:

- from — is a LOGICAL*1 array variable, or a Hollerith literal, giving the numeric characters to be converted.
- integer — is an INTEGER variable which will be set to the integer resulting from the conversion.
- cnumb — is an INTEGER variable which, on entry to DTB, should contain the maximum number of characters to be scanned in the conversion. On exit from DTB, cnumb is set to the actual number of characters scanned.
- dnumb — is an INTEGER variable which will be set to the number of significant digits in integer. The sign is not included in this number.
- fill — is a LOGICAL*1 character variable, or a Hollerith literal, specifying a character to be ignored if it precedes the numeric digits in the string.
- err — (optional) is the number of a FORTRAN statement to transfer to if invalid characters or multiple signs are encountered, if the converted number is too large to hold in a FORTRAN fullword INTEGER, or if, on entry, cnumb≤0.

Comments: A single sign (+ or -) may be imbedded in the leading fill characters and will determine the sign of integer. If there is no sign, '+' is assumed.

DTB can be used to reverse any action of the BTD subroutine.

If the field from is all fill characters, then integer and dnumb are set to zero. If the field from is all zeros, then integer is set to zero and dnumb is set to cnumb, the actual number of zeros in the field.

If the error return to statement err is taken because of invalid characters or adjacent multiple signs, then integer=dnumb=0 and cnumb is set to the number of characters scanned before the error was encountered.

There will be no error return taken once a digit is encountered. After the first digit, any nondigit (even another sign or a fill character) terminates the number.

If the error return to statement err is taken because the converted number was too large to hold in the fullword integer, then integer=0, dnumb is set to the number of digits encountered, and cnumb is set to the total number of characters in the field (fill characters plus sign character plus numeric characters).

If the error return to statement err is taken because cnumb≤0, then integer=dnumb=0 and cnumb remains unchanged.

Example: The example below converts the character string

```
.....-139.....
```

stored starting in element 30 of array NUMB, into an integer number:

```
LOGICAL*1 NUMB(75)
NC=14
CALL DTB(NUMB(30),I,NC,ND,'.',&10)
```

On exit, I=-139, NC=9, and ND=3.

October 1983

EQUC

Purpose: To compare two characters for equality.

Calling Sequence:

FORTRAN:
```
LOGICAL EQUC
IF (EQUC(char1,char2)) statement
```

Parameters:

char1,char2 are LOGICAL*1 variables or array elements, or single-character Hollerith literals, to be compared for equality.

statement is a FORTRAN statement to transfer to if char1 and char2 are equal.

Comment: If char1 is identical to char2, then EQUC(char1,char2) has the value .TRUE.; otherwise, it has the value .FALSE.

Example: The example below transfers to statement number 10 if the 7th element of ARRAY is the letter G.

```
LOGICAL EQUC
LOGICAL*1 ARRAY(25)
IF (EQUC('G',ARRAY(7))) GO TO 10
```

FINDC

Purpose: To search for any one of a set of characters.

Calling Sequence:

FORTRAN: CALL FINDC(array,len,char,numb,start,finish, cfound,&err1,&err2)

Parameters:

- array is the LOGICAL*1 character array to be searched.
- len is an INTEGER expression giving the position in array of the last character to be searched.
- char is either an array variable indicating the characters for which to search or a Hollerith literal specifying the characters.
- numb is an INTEGER expression giving the number of characters in char. If numb=0, then the same characters as given in a preceding call with numb>0 will be used.
- start is an INTEGER expression indicating the position in array at which the search is to start.
- finish is an INTEGER variable which will contain the position in array at which a character in char is found. If none of the characters is found, finish is set to zero.
- cfound is an INTEGER variable which will be set to the position in char of the character which is found. If none of the characters is found, cfound is set to zero.
- err1 (optional) is the number of a FORTRAN statement to transfer to if none of the characters is found in the search.
- err2 (optional) is the number of a FORTRAN statement to transfer to if start≤0, start>len, or numb<0.

Comment: If numb=0 on the first call to FINDC, no characters will be found. Control will be transferred to the statement numbered err2.

Example: The example below searches the array LARRAY for the first occurrence of the numeric characters 0,1,2,3,...,9.

```
LOGICAL*1 LARRAY(125)
CALL FINDC(LARRAY,125,'0123456789',10,1,IF,ICF,&10)
```

If LARRAY contains the character '7' in position 39, i.e., in LARRAY(39), with no numeric characters preceding it, then, upon exit from FINDC, IF will be 39 and ICF will be 8, indicating that the 8th character in the string '0123456789' was found in LARRAY(39). If there are no numeric characters in LARRAY, then control will transfer to statement 10 with IF=ICF=0.

If, on subsequent calls to FINDC, the same characters 0,1,2,3,...,9 are to be searched for, then the fourth parameter numb should be set to zero so that initialization need not be repeated.

FINDST

Purpose: To search an array for a specified character string.

Calling Sequence:

FORTRAN: CALL FINDST(array,len,string,numb,start,finish, &err1,&err2)

Parameters:

- array is the LOGICAL*1 character array to be searched.
- len is an INTEGER expression giving the position in array of the last character in the search.
- string is an array variable, or a Hollerith literal, indicating the character string for which to search.
- numb is an INTEGER expression giving the number of characters in string.
- start is an INTEGER expression indicating the position in array at which the search is to start.
- finish is an INTEGER variable which will be set to the position of the character in array at which string starts. If string is not found, finish is set to zero.
- err1 (optional) is the number of a FORTRAN statement to transfer to if string is not found.
- err2 (optional) is the number of a FORTRAN statement to transfer to if start≤0, start>len, or numb≤0.

Comment: The complete string must be within the limits start and len of array.

Example: The example below searches the array AR for the string MODE with the search starting at the 10th character and continuing to the 40th character.

```
LOGICAL*1 AR(50)
CALL FINDST(AR,40,'MODE',4,10,IFINIS,&12)
```

IGC

Purpose: To ignore all of a set of characters, i.e., to find the first character which is not one of a specified set of characters.

Calling Sequence:

FORTRAN: CALL IGC(array,len,char,numb,start,finish, &err1,&err2)

Parameters:

- array is the LOGICAL*1 character array to be searched.
- len is an INTEGER expression giving the position in array of the last character in the search.
- char is either an array variable containing, or a Hollerith literal specifying, the characters to be ignored.
- numb is an INTEGER expression giving the number of characters in char. If numb=0, the characters given in a preceding call with numb>0 will be used in the search.
- start is an INTEGER expression giving the position in array of the character at which the search is to start.
- finish is an INTEGER variable which will be set to the character position in array at which the first character different from those in char is found. If all characters are ignored, finish is set to zero.
- err1 (optional) is the number of a FORTRAN statement to transfer to if all characters are ignored.
- err2 (optional) is the number of a FORTRAN statement to transfer to if start≤0, start>len, or numb<0.

Comment: If numb=0 on the first call to IGC, no characters are ignored; finish is set equal to start.

Example: The example below searches for the first nonblank character in the array LARRAY.

```
LOGICAL*1 LARRAY(212)
CALL IGC(LARRAY,212,' ',1,1,IF,&10)
```

If the first nonblank character is in character position 132 of the array, IF will be set to 132. If all

characters are blank, then IF will be set to zero and control will transfer to statement number 10.

October 1983

LCOMC

Purpose: To determine whether one character string is less than, equal to, or greater than another string.

Calling Sequence:

FORTRAN: i=LCOMC(numb,string1,string2)

Parameters:

numb is an INTEGER expression giving the number of characters in each string.

string1,string2 are the character strings to be compared for equality. They may be specified either by an array variable or by a Hollerith literal. Equality is interpreted in the sense of position within the 360 collating sequence.

Values Returned:

LCOMC is a FUNCTION subprogram and will return an integer i having a value of:

+1 if string1>string2, i.e., if string1 follows string2 in the collating sequence.
0 if string1=string2, i.e., if the character strings are identical.
-1 if string1<string2, i.e., if string1 precedes string2 in the collating sequence.

Comment: If numb≤0, no comparison is made and i is set to zero.

Example: The example below compares 2 character strings of 20 characters starting at A(1) and B(19) and branches to statement 12 on equality.

```
LOGICAL*1 A(50),B(60)
IF(LCOMC(20,A(1),B(19)).EQ.0) GO TO 12
```

MOVEC

Purpose: To move character strings from one place to another.

Calling Sequence:

FORTRAN: CALL MOVEC(numb,from,to,&err)

Parameters:

- numb — is an INTEGER expression giving the number of characters to be moved. numb must be greater than zero.
- from — is either an array variable containing the character string to be moved or a Hollerith literal specifying the string.
- to — is an array variable indicating the start of the place to which the from characters are to be moved.
- err — (optional) is the number of a FORTRAN statement to transfer to if numb≤0 or numb>32767.

Comments: The from and to array variables can indicate portions of the same array. In fact, they can be overlapping portions. However, in the latter case, the user must ensure that characters to be moved are not replaced before being moved. The characters are moved one at a time from the first to the numbth position.

If numb≤0 or numb>32767, no transfer of characters will occur.

Example: The example below moves 7 characters, starting with the 10th character of array AR1, to AR2, starting with the 80th character.

```
LOGICAL*1 AR1(100),AR2(132)
CALL MOVEC(7,AR1(10),AR2(80))
```

The example below moves the character string ERROR MESSAGES into the array MSG.

```
LOGICAL*1 MSG(80)
CALL MOVEC(14,'ERROR MESSAGES',MSG)
```

The example below moves the 4 characters DATA into a simple INTEGER variable I.

```
DATA X/'DATA'/
CALL MOVEC(4,X,I)
```

October 1983

SETC

Purpose: To set adjacent characters equal to a specified character.

Calling Sequence:

FORTRAN: CALL SETC(numb,array,char,&err)

Parameters:

numb	is an INTEGER expression giving the number of characters to be set.
array	is an array variable giving the starting position of the characters to be set.
char	is either a variable containing the character to which the numb characters are to be set or a Hollerith literal specifying the character.
err	(optional) is the number of a FORTRAN statement to transfer to if numb≤0.

Comment: If numb≤0, no characters are changed.

Example: The example below sets all of the characters in the array A to blanks.

```
LOGICAL*1 A(50)
CALL SETC(50,A,' ')
```

TRNC

Purpose: To translate specified characters in an array into other characters.

Calling Sequence:

FORTRAN: CALL TRNC(numb,array,oldchar,newchar,cnumb,&err)

Parameters:

- numb is an INTEGER expression giving the number of characters for translation.
- array is an array variable giving the starting position of the characters for translation.
- oldchar is either an array variable containing a list of the characters to be translated, or a Hollerith literal specifying the characters.
- newchar is either an array variable containing a list of the characters into which oldchar is to be translated, or a Hollerith literal specifying the characters. Any occurrence of the first character in oldchar will be translated into the first character of newchar, the second character of oldchar into the second of newchar, etc.
- cnumb is an INTEGER expression giving the number of characters in oldchar and newchar. If cnumb=0, then oldchar and newchar as given in a preceding call with cnumb>0 will be used.
- err (optional) is the number of a FORTRAN statement to transfer to if numb≤0 or cnumb<0.

Comments: The routine does not check for duplication of characters in oldchar. The final appearance of a duplicated character will dictate its translation.

It is the user's responsibility to ensure that there are the same number of characters in oldchar and newchar. If there are not, unpredictable translations may occur.

If numb≤0 or cnumb<0 (or ≤0 on the first call), no translation will occur. All characters not mentioned in oldchar are left alone.

Example: The example below translates all As to 1s, Bs to 2s, and Cs to 3s in the array CHAR.

```
LOGICAL*1 CHAR(65)
CALL TRNC(65,CHAR,'ABC','123',3)
```

October 1983

TRNST

Purpose: To search for a given character string and translate it into another string.

Calling Sequence:

FORTRAN: CALL TRNST(array,len,oldst,newst,numb,start, finish,&err1,&err2)

Parameters:

- array is the LOGICAL*1 character array to be searched.
- len is an INTEGER expression giving the character position in array at which searching is to terminate.
- oldst is either an array variable containing the character string to be translated or a Hollerith literal specifying the character string.
- newst is either an array variable containing the new character string or a Hollerith literal specifying the string.
- numb is an INTEGER expression giving the number of characters in the strings.
- start is an INTEGER expression giving the position in array at which searching is to start.
- finish is an INTEGER variable which will be set to the starting position of the translated string. finish will be set to zero if the string is not found.
- err1 (optional) is the number of a FORTRAN statement to transfer to if oldst is not found in the search.
- err2 (optional) is the number of a FORTRAN statement to transfer to if start≤0, start>len, or numb≤0.

Comments: oldst and newst must be the same lengths. Only the first occurrence of oldst is translated. oldst must be completely within the limits start and len of array for translation to occur.

Example: The example below translates the string RECIEVE in the array A to RECEIVE.

```
LOGICAL*1 A(200)
CALL TRNST(A,200,'RECIEVE','RECEIVE',7,1,IF,&30)
```

If the string is found starting in character 29 of A, then IF will be set to 29. If the string is not found, then IF=0 and control is transferred to statement number 30.

October 1983

LOGICAL OPERATORS

The logical operators package makes the following 360 machine instructions directly available to the FORTRAN user: MVC, CLC, NC, OC, XC, TR, TRT, ED, and EDMK. The package resides in *LIBRARY.

The package has the following entry points: IMVC, ICLC, INC, IOC, IXC, ITR, ITRT, IED, and IEDMK.

Calling Sequences:

```
I = IMVC(len,base1,displ1,base2,displ2)
I = ICLC(len,base1,displ1,base2,displ2)
I = INC(len,base1,displ1,base2,displ2)
I = IOC(len,base1,displ1,base2,displ2)
I = IXC(len,base1,displ1,base2,displ2)
I = ITR(len,base1,displ1,base2,displ2)
I = ITRT(len,base1,displ1,base2,displ2,dr,fb)
I = IED(len,base1,displ1,base2,displ2)
I = IEDMK(len,base1,displ1,base2,displ2,dr)
```

Parameters:

len	is the integer length in bytes. No restriction is placed on the size of len. An error message will be generated if len < 0; or, for the entries IED or IEDMK, if len > 256.
base1	is the base location of the first operand.
displ1	is the integer displacement in bytes for the first operand. No restriction is placed on the size of displ1.
base2	is the base location of the second operand.
displ2	is the integer displacement in bytes for the second operand. No restriction is placed on the size of displ2.
dr	is an integer return parameter for ITRT and IEDMK only. For ITRT, dr will contain the displacement in bytes from the beginning of the argument list, (base1+displ1), to the argument corresponding to the first nonzero function byte (if any). For IEDMK, dr will contain the displacement in bytes from the beginning of the source, (base2+displ2), to the result character, whenever the latter is a zoned source digit and the significance indicator was off before the examination. In both cases, dr will be set to zero if the resulting condition code is zero.
fb	is an optional integer return parameter for ITRT. When a nonzero function byte is found, it will be returned in fb as an integer in the range (0,255); otherwise, fb will be zero.

For the complete description of the machine instructions, see the IBM publication, IBM System/370 Principles of Operation, form GA22-7000. These subroutines are coded as integer-valued functions with the resulting condition code (0, 1, or 2) as the value.

In the abbreviated descriptions below, the first operand consists of len bytes beginning at location base1+displ1, and the second operand consists of len bytes beginning at location base2+displ2. These two operands may overlap in any manner. For all five of these entry points, processing is carried out left to right one byte at a time. Note that the result of performing an operation on the first bytes of the two operands is stored before the second bytes are fetched so that overlap can have a significant effect on the result.

IMVC - Move the second operand into the first operand location.
INC - Replace the first operand by the logical product (AND) of the operands.
IOC - Replace the first operand by the logical sum (OR) of the operands.
IXC - Replace the first operand by the modulo-two sum (exclusive OR) of the two operands.
ICLC - Compare the two operands. The operation is terminated as soon as two unequal bytes are found.

The result of an IMVC is always zero. The result of an INC, IOC, or IXC is zero if the result operand is zero, and one, otherwise. The result of an ICLC is 0, 1, or 2, depending on whether the first operand is equal to, less than, or greater than the second operand.

For the ITR and ITRT entries, the first operand consists of len bytes beginning at location base1+displ1, and the second operand consists of a 256-byte function table beginning at location base2+displ2. These operands may overlap, but probably not too fruitfully. The ITR entry translates each byte of the first operand by replacing it with the corresponding byte from the function table. The result of an ITR operation is always zero. The ITRT entry does not change either operand. Processing the first operand bytes left to right, the corresponding function byte is interrogated. If the function byte is zero, the processing of the first operand continues. If the function byte is nonzero, the operation is terminated. When terminated, processing is terminated with the byte at location base1+displ1+dr and the corresponding nonzero function byte is available in fb. The result of the ITRT will be 1 if this byte is not the last byte of the first operand, and 2 if it is the last byte. If no nonzero function byte is encountered, the result of an ITRT will be zero, and dr and fb will be indeterminate.

The complexity of the IED and IEDMK instructions precludes any short descriptions here.

The following examples illustrate the use of the logical operators.

```
      INTEGER A,B
      B = 31
      LEN = 4
      IR = INC(LEN,A,0,B,0)
```

The logical AND product of A and B will replace A. In this case, B = 31 so A will be replaced by (A mod 32). IR will be set to 0 or 1 depending on whether the result in A is zero or nonzero.

```
      INTEGER A(4),B(4),D1,D2
      READ 2, (A(I),I=1,4), (B(I),I=1,4)
2     FORMAT(4A4)
      D1 = 8
      D2 = 0
      IR = ICLC(8,A,D1,B,D2)
```

This program logically compares the string in A(3), A(4), to the string in B(1), B(2). IR will be set to 0, 1, or 2 depending on whether the first string is equal to, less than, or greater than the second string.

October 1983

October 1983

BITWISE LOGICAL FUNCTIONS

These simple functions do the bitwise logical operations which are difficult to state in FORTRAN arithmetic formulas. If their names are prefixed with an "L", they are integer; otherwise, they are declared real. The only exception to this rule is that SHFTR and SHFTL must be declared integer. This package resides in *LIBRARY.

The functions available are: AND, LAND, OR, LOR, XOR, LXOR, COMPL, LCOMPL, SHFTR, and SHFTL.

Calling Sequences:

```
 AND      C = AND(A,B)
LAND     IC = LAND(IA,IB)
```

The result has bits on only if the corresponding bits of the arguments are both on.

```
 OR       C = OR(A,B)
LOR      IC = LOR(IA,IB)
```

The result has bits on only if either or both arguments have the corresponding bits on.

```
 XOR      C = XOR(A,B)
LXOR     IC = LXOR(IA,IB)
```

The result has bits on only if the corresponding bits of the two arguments are not the same.

```
 COMPL    B = COMPL(A)
LCOMPL   IB = LCOMPL(IA)
```

The result has all the bits of the argument reversed.

```
SHFTR    IC = SHFTR(IA,IB)
SHFTL    IC = SHFTL(IA,IB)
```

The first argument is shifted right or left by the number of bits specified by the last 6 bits of the second integer argument (i.e., modulo 64). As logical shift functions, they are not equivalent to a division or to a multiplication by a power of two.

Unless otherwise stated, the arguments of the functions may be either real or integer provided that they are fullwords (four bytes long).

The functions LAND, LOR, LXOR, LCOMPL, SHFTR and SHFTL may be generated as in-line code by the FORTRAN-H compiler by specifying the XL option. See the section "*FTN Interface" in this volume for details.

The following examples illustrate the use of the bitwise logical functions.

```
          WORD = XOR(WORD,WORD)
```

This example zeros all the bits of the fullword WORD.

```
          DATA MASK/Z00FF0000/
          SCDBYT = AND(WORD,MASK)
```

This example examines the second byte of the fullword WORD by deleting the other bytes and storing the result into the fullword SCDBYT.

```
          IWORD = SHFTR(IWORD,24)
```

This example moves the first byte of the fullword IWORD into the fourth byte position and leaves the other bytes zero.

```
                READ (5,4) (CHAR(I),I=1,4)
          4     FORMAT(4A1)
                DATA MASK/ZFF000000/
                WORD = 0.
                DO 6 I=1,4
          6     WORD = OR(WORD,SHFTR(AND(CHAR(I),MASK),(I-1)*8))
```

This example packs four characters into one word.

October 1983

BMS (BIT MANIPULATION SUBROUTINES)

BMS is a subroutine package that enables the user to manipulate bit strings. It was written with the FORTRAN user in mind, so most examples are in FORTRAN. However, these subroutines may be called from any program that uses the standard OS type I (S-type) calling conventions that FORTRAN uses; a few examples are included to illustrate this.

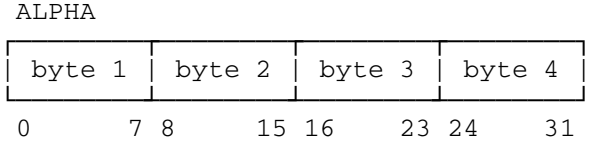
A bit string is a region of contiguous bits in the user's storage. It need not begin or end on any of the recognized storage boundaries. To define a bit string to a BMS subroutine, the user passes three parameters: baseadd, bitdisp, and bitlen.

baseadd	is a valid address in the user's storage.
bitdisp	is a fullword integer containing a displacement in bits from baseadd (may be 0 or a positive integer).
bitlen	is a fullword integer containing the length of the string in bits (may be 0 or a positive integer).

baseadd and bitdisp together determine the beginning of the string in a manner analogous to a base address and a displacement in a 360/370 machine instruction, the difference being that bitdisp is a displacement in bits rather than bytes. For example,

```
baseadd = ALPHA, a fullword variable
bitdisp = 16
bitlen  = 8
```

The bit string defined is the third byte of ALPHA.



The subroutines are of two types: subroutines and integer-valued functions. The subroutines all have a normal return and an error return. Since they all work the same way, the return codes are summarized here:

Return Codes:

0 Operation successful.
4 Negative parameter passed or wrong number of parameters passed.

FORTRAN users can take advantage of the return code by coding an ampersand followed by a statement number after the last parameter of a subroutine; if the return code is 4, the subroutine will return to the specified statement, rather than to the point from which the subroutine was called.

The subroutines available in the BMS package are:

Subroutine	Function
BCLEAR	Clear a bit string to zeros
BSET	Set a bit string to ones
BFLIP	Complement a bit string (NOT)
BCOPY	Copy a bit string to another location in storage
BSWAP	Switch 2 bit strings in storage
BAND	Calculate the logical product (AND) of 2 bit strings
BOR	Calculate the logical sum (OR) of 2 bit strings
BXOR	Calculate the modulo-two sum (XOR) of 2 bit strings
BFETCH	Return a bit string as an integer value
BCOMP	Compare 2 bit strings (<, =, >)
BOOLE	Perform on 2 bit strings the boolean operation defined by a truth table passed as an argument
BINSRT	Insert a substring in a bit string
BDLETE	Delete a substring from a bit string
BSCAN	Find the location in a bit string of a substring
BCOUNT	Count the occurrences of a substring in a bit string

October 1983

BCLEAR

Purpose: To clear a bit string to zeros.

Location: *LIBRARY

Calling Sequence:

```
CALL BCLEAR(baseadd,bitdisp,bitlen[,&err])
```

Examples:

```
CALL BCLEAR(A(I),0,16)
```

The halfword beginning at A(I) is cleared.

```
CALL BCLEAR(B(J),9,99)
```

99 bits beginning with the 10th bit (i.e., bit 9) of B(J) are cleared.

```
INTEGER*4 X
CALL BCLEAR(X,32-N,N,&99)
```

When 0 ≤ N ≤ 32, the N low-order bits of the fullword X are cleared; otherwise, control passes to statement 99.

```
Var Disp, Length:Integer;
Var Alpha : Array[1..100] of Integer;
Procedure Bclear(Var Baseadd,Bitdisp,Bitlen : Integer);
          Fortran;
   ...
Disp   := 0;
Length := 3200;
   ...
Bclear(Alpha,Disp,Length);
```

In above Pascal example, the array Alpha is cleared to zeros.

```
Var Disp, Length K : Integer;
Procedure Bclear(Var Baseadd,Bitdisp,Bitlen : Integer);
          Fortran;
   ...
Disp   := 12;
Length := 1;
   ...
Bclear(K,Disp,Length);
```

In above Pascal example, bit 12 of K is cleared.

```
DCL I FIXED BINARY(31),
    BCLEAR EXTERNAL ENTRY(FIXED BINARY(31),
                          FIXED BINARY(31),
                          FIXED BINARY(31));
   ...
CALL BCLEAR(I,8,1)
```

In above PL/I Optimizer example, bit 8 of I is cleared.

October 1983

BSET

Purpose: To set a bit string to ones.

Location: *LIBRARY

Calling Sequence:

```
CALL BSET(baseadd,bitdisp,bitlen[,&err])
```

Description: This subroutine works like BCLEAR.

BFLIP

Purpose: To complement a bit string (all 1s in the string become 0s; all 0s become 1s).

Location: *LIBRARY

Calling Sequence:

```
CALL BFLIP(baseadd,bitdisp,bitlen[,&err])
```

Description: This subroutine works like BCLEAR.

BCOPY

Purpose: To copy a bit string to another place in storage.

Location: *LIBRARY

Calling Sequence:

```
CALL BCOPY(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
     [,&err])
```

Description: The parameters specify two bit strings of equal length. The contents of the first string are copied to the storage occupied by the second string, destroying the previous contents of the second string. The contents of the first string are unchanged.

Examples:

```
INTEGER*4 RATE,CODES(100)
CALL BCOPY(RATE,29,3,CODES(I),14)
```

This copies a 3 bit code from the low-order positions of the fullword integer RATE into bits 14-16 of the _i_th entry of a packed table of codes.

```
CALL BCOPY(CODES(I),14,3,RATE,29)
```

This performs the reverse operation, putting the code back in RATE. Note that BCOPY does not clear the high-order positions of RATE. To unpack the bit string (i.e., to transform it to an integer), set RATE to zero before calling BCOPY.

October 1983

BSWAP

Purpose: To switch two bit strings in storage.

Location: *LIBRARY

Calling Sequence:

```
CALL BSWAP(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
     [,&err])
```

Description: The parameters specify two bit strings of equal length. These two strings are switched in storage.

Examples:

```
INTEGER*2 A(500)
CALL BSWAP(A(I),0,8,A(J),8)
```

The high-order byte of A(I) is swapped with the low-order byte of A(J).

BAND

Purpose: To calculate the logical product (AND) of two bit strings.

Location: *LIBRARY

Calling Sequence:

```
CALL BAND(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
     [,baseadd3,bitdisp3,&err])
```

Description: The parameters specify three bit strings of equal length. The contents of the first and second strings are ANDed and the result is stored in the third string. If the third string is omitted, the result is stored in the second string.

Examples:

```
INTEGER P(10),Q(10),R(10)
CALL BAND(P(I),0,4,Q(J),28,R(K),3)
```

The 4 high-order bits of P(I) are ANDed with the 4 low-order bits of Q(J) and the result is stored in bits 3-6 of R(K).

```
CALL BAND(A,0,8,B,0)
```

The first byte of A is ANDed with the first byte of B and the result is stored in the first byte of B.

```
INTEGER*2 MASK/Z03FF/
CALL BAND(MASK,0,16,K,0)
```

The 6 high-order bits of K are cleared. The same result can be obtained by CALL BCLEAR (K,0,6).

```
Var Disp1, Disp2, Disp3, Length, I, J, K : Integer;
Procedure Band(Var Baseadd,Bitdisp,Bitlen,Baseadd2,
              Bitdisp2,Baseadd3,Bitdisp3); Fortran;
   ...
Disp1 := 0; Disp2 := 0; Disp3 := 3; Length := 32;
   ...
Band(I,Disp1,Length,J,Disp2,K,Disp3);
```

In above Pascal example, all 32 bits of I and J are ANDed; the result is stored in K.

BOR

Purpose: To calculate the logical sum (OR) of two bit strings.

Location: *LIBRARY

Calling Sequence:

```
CALL BOR(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
     [,baseadd3,bitdisp3,&err])
```

Description: This subroutine works like BAND.

BXOR

Purpose: To calculate the modulo-two sum (XOR) of two bit strings.

Location: *LIBRARY

Calling Sequence:

```
CALL BXOR(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
     [,baseadd3,bitdisp3,&err])
```

Description: This subroutine works like BAND.

BFETCH

Purpose: To return a bit string as an integer value.

Location: *LIBRARY

Calling Sequence:

```
ivar = BFETCH(baseadd,bitdisp,bitlen)
```

Description: If the bit string specified is between 1 and 31 bits long, it is returned as a positive integer or zero. If bitdisp is negative or if bitlen is not in range, a value of -1 is returned.

Examples:

```
INTEGER BFETCH
RATE = BFETCH(CODES(I),14,3)
```

This does the same unpacking job as the second example of BCOPY, except that it also clears the high-order bits of RATE. It is faster than BCOPY.

```
INTEGER BFETCH
IF (BFETCH(X,Y,Z).EQ.KEY) GO TO 99
```

A bit string is compared to another integer.

```
Var Source, Disp, Length, I : Integer;
Function Bfetch(Var Baseadd,Bitdisp,Bitlen) : Integer;
         Fortran;
   ...
Disp   := 20;
Length := 5;
   ...
I := Bfetch(Source,Disp,Length);
```

In above Pascal example, the five bits of SOURCE beginning with bit 20 are returned to I.

October 1983

BCOMP

Purpose: To compare two bit strings.

Location: *LIBRARY

Calling Sequence:

```
ivar = BCOMP(baseadd,bitdisp,bitlen,baseadd2,bitdisp2
       [,bitlen2])
```

Description: The first string (specified by baseadd, bitdisp, and bitlen) is compared with the second string (specified by baseadd2, bitdisp2, and bitlen2). If bitlen2 is omitted, it is taken to be equal to bitlen. The strings are compared bit for bit and from left to right, until a difference occurs, or until one or both of the strings is exhausted. If a difference occurs, the string with zero in the position of difference is considered to be the lesser of the two strings (e.g., 0110 < 10). If the strings are equal until one of the strings is exhausted, then the shorter string is considered to be the lesser of the two (e.g., 01 < 0110). Two strings are considered equal only if they have both the same contents and the same length (e.g., 0110 = 0110).

Function value returned:

```
BCOMP = -1, if string1 < string2
BCOMP =  0, if string1 = string2
BCOMP =  1, if string1 > string2
BCOMP =  2, if negative parameter is passed or wrong
            number of parameters is passed.
```

Example:

```
INTEGER BCOMP
IF (BCOMP(INPUT,INPTR,24,'YES',0).EQ.0) GO TO 12
```

If the 24-bit (i.e., 3-character) string in INPUT is equal to the character string "YES", then go to statement 12.

BOOLE

Purpose: To perform on two bit strings the boolean function specified by a truth table.

Location: *LIBRARY

Calling Sequence:

```
CALL BOOLE(baseadd,bitdisp,bitlen,baseadd2,bitdisp2,
     tbase,tdisp[,baseadd3,bitdisp3][,&err])
```

Description: The parameters specify three bit strings of equal length, and a truth table four bits long. The contents of the first and second strings are operated upon as described in the truth table and the result is stored in the third string. If the third string is omitted, the result is stored in the second string.

The truth table consists of four bits; the first bit contains the result if a bit in the first string and the corresponding bit in the second string are both ones; the second bit indicates the result if the bit in the first string is one and the bit in the second string is zero; the third bit indicates the result if the first bit is zero and the second is one; and the third contains the result if both bits are zero.

Example:

```
I = 3
J = 5
K = 3
CALL BOOLE(I,28,4,J,28,K,28,L,28)
```

In this sequence, two bit strings, 0101 and 1001, are operated upon by the truth table 0101. The result 0110 is placed in the last four bits of integer K.

A truth table containing 1000 is equivalent to AND.

A truth table containing 1110 is equivalent to inclusive OR.

A truth table containing 0110 is equivalent to exclusive OR.

October 1983

BINSRT

Purpose: To insert a substring into a bit string.

Location: *LIBRARY

Calling Sequence:

```
CALL BINSRT(baseadd,bitdisp,bitlen,baseadd2,bitdisp2,
     bitlen2,where[&err])
```

Description: The second string is copied into the first, starting at bit number where. The bits running from where to bitlen-1 in the first string are shifted rightward bitlen2 places to accommodate the insertion of the second string.

No storage management is done; it is the caller's responsibility to ensure that expansion of the first string is permissible.

Example:

```
I = 124
J = 0
CALL BINSRT(I,25,5,J,30,2,3)
```

This sequence inserts the string 00 into the string 11111, starting at position 3; the result is 1100111, which is stored starting with bit 25 of I.

BDLETE

Purpose: To delete a substring from a bit string.

Location: *LIBRARY

Calling Sequence:

```
CALL BDLETE(baseadd,bitdisp,bitlen,bitdisp2,bitlen2
     [,&err])
```

Description: The triplets baseadd, bitdisp, bitlen, and baseadd, bitdisp2, bitlen2 describe two bit strings, the second of which must be contained in the first (i.e., bitdisp2 must be greater than or equal to bitdisp, and bitdisp2+bitlen2 must be less than or equal to bitdisp+bitlen). The second string is deleted from the first; in effect, the bits running from baseadd+bitdisp2+bitlen2 to baseadd+bitdisp+bitlen are copied to baseadd+bitdisp2. The length of the string is effectively decreased from bitlen to bitlen-bitlen2; the contents of the (now) unused bit positions after the end of the new string are undefined. No storage management is done.

Example:

```
I = 14
CALL BDLETE(I,25,7,28,3)
```

This sequence deletes three bits, starting at location 3 (28-25=3) from the bit string 00011100; the result 0000 is stored starting with bit 25 of I.

October 1983

BSCAN

Purpose: To find the location of a substring in a bit string.

Location: *LIBRARY

Calling Sequence:

```
INTEGER BSCAN
i = BSCAN(baseadd,bitdisp,bitlen,baseadd2,bitdisp2,
    bitlen2)
```

Description: The parameters specify two bit strings, the second of which should be shorter than the first. The value returned is the offset from baseadd of the point (within the first string) at which the second string is to be found. If the second string is not found within the first, -2 is returned. If any errors are detected in the parameters, the return value is -1.

Example: Suppose a bit string 00101010 starting at bit 24 of J, and bit string 101 starting at bit 29 of K, where J and K are both integers; then

```
INTEGER BSCAN
I = BSCAN(J,24,7,K,29,3)
```

returns the value 26 to I.

BCOUNT

Purpose: To count the number of occurrences of a substring in a bit string.

Location: *LIBRARY

Calling Sequence:

```
INTEGER BCOUNT
i = BCOUNT(baseadd,bitdisp,bitlen,baseadd2,bitdisp2,
    bitlen2)
```

Description: The parameters specify two bit strings, the second shorter than the first. The returned value indicates the number of copies of the second string to be found in the first. If any error is found in the parameters, -1 is returned.

Example: Suppose a bit string 10101010 starting at bit 24 of J and bit string 01 starting at bit 30 of K, where J and K are both integers. Then

```
INTEGER BCOUNT
I = BCOUNT(J,24,8,K,30,2)
```

returns the value 3 to I.

October 1983

ANSI STANDARD BIT MANIPULATION SUBROUTINES

This set of subroutines contains procedures for bit manipulation with integers and date/time functions as described in ANSI/ISA-S61.1, Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output, and Bit Manipulation, as well as additional bit manipulation functions as described in Military Standard 1753, FORTRAN, DOD Supplement to American National Standard X3.9-1978. Other subroutines described in ANSI/ISA-S61.1, the executive interface and the process input/output function interfaces, do not apply to the MTS environment and thus are not implemented.

These subroutines are intended to allow FORTRAN programs written for other systems that provide subroutines implementing the same standards to be run in MTS with little or no modification, and to facilitate the development in MTS of FORTRAN programs intended for use on such systems.

The following subroutines are available:

Subroutine	Function
IOR	Inclusive OR of the bits in two integers.
IAND	Logical AND of two integers.
IEOR	Exclusive OR of two integers.
NOT	Logical complement of an integer.
ISHFT	Shift bits right or left (noncircular).
BTEST	Test a specific bit.
IBSET	Set a bit to one.
IBCLR	Clear a bit to zero.
ISHFTC	Circular shift of some or all of the bits in an integer.
IBITS	Extract a bit substring.
MVBITS	Move bits from one integer to another.
DATE	Return current date.
ANSITM	Return current time.

The ANSITM subroutine is named TIME in the standard. However, since there is a different MTS subroutine named TIME, a different name had to be chosen for the ANSI subroutine. The object-file editor can be used to change calls to TIME to calls to ANSITM (see the ANSITM description for an example).

Although these subroutines were intended for FORTRAN programs in the standard, they may be called from any programming language that uses the standard IBM OS S-type linkage conventions.

IOR

Purpose: To perform an inclusive OR on the bits comprising two fullword integers.

Location: *LIBRARY

Calling Sequence:

```
i = IOR(j,k)
```

Parameters:

j,k is a fullword (INTEGER*4) integers.
i is a fullword integer to receive the inclusive OR of j and k.

Description: Each bit in i is set to one, if the corresponding bit in j or k or in both is equal to one. If the corresponding bits in both j and k are equal to zero, then the bit in i is set to zero.

Example:

```
      J = 5
      K = 3
C
C In binary,
C J = 00000000 00000000 00000000 00000101
C K = 00000000 00000000 00000000 00000011
C
      I = IOR(J,K)
C
C The value returned to I is 7; in binary,
C I = 00000000 00000000 00000000 00000111.
C
```

October 1983

IAND

Purpose: To perform a logical AND on the bits comprising two fullword integers.

Location: *LIBRARY

Calling Sequence:

```
i = IAND(j,k)
```

Parameters:

j,k is a fullword (INTEGER*4) integers.
i is a fullword integer containing the logical AND of j and k.

Description: Each bit in i is set to one, if the corresponding bits in both j and k are equal to one.

Example:

```
      J = 3
      K = 5
C
C In binary,
C J = 00000000 00000000 00000000 00000011
C K = 00000000 00000000 00000000 00000101
C
      I = IAND(J,K)
C
C The value returned to I is 1; in binary,
C I = 00000000 00000000 00000000 00000001.
C
```

IEOR

Purpose: To form the exclusive OR of two fullword integers.

Location: *LIBRARY

Calling Sequence:

```
i = IEOR(j,k)
```

Parameters:

j,k	is a fullword (INTEGER*4) integers.
i	is a fullword integer containing the exclusive OR of j and k.

Description: Each bit in i is set to one, if the corresponding bit in either j or k (but not both) is equal to one.

Example:

```
      J = 3
      K = 5
C
C In binary,
C J = 00000000 00000000 00000000 00000011
C K = 00000000 00000000 00000000 00000101
C
      I = IEOR(J,K)
C
C The value returned to I is 6; in binary,
C I = 00000000 00000000 00000000 00000110.
C
```

October 1983

NOT

Purpose: To return the logical complement of a fullword integer.

Location: *LIBRARY

Calling Sequence:

```
i = NOT(j)
```

Parameters:

j	is a fullword (INTEGER*4) integer.
i	is a fullword integer containing the complement of j.

Description: Each bit in i is set to one, if the corresponding bit in j is equal to zero.

Example:

```
      J = -1
C
C In binary,
C J = 11111111 11111111 11111111 11111111
C
      I = NOT(J)
C
C The value returned to I is 0; in binary,
C I = 00000000 00000000 00000000 00000000.
C
```

ISHFT

Purpose: To shift the bits comprising an integer to the right or left.

Location: *LIBRARY

Calling Sequence:

```
i = ISHFT(j,k)
```

Parameters:

- j,k is a fullword (INTEGER*4) integers.
- j is a fullword integer to be shifted.
- k is a fullword integer containing the number of positions that j is to be shifted.
- i is a fullword integer containing the value of j shifted k positions.

Description: Each bit in j is shifted to the left by k positions if k is positive, and to the right by k positions if k if negative.

If k is greater than 32 or less than -32, an error message is produced and execution is terminated.

Example:

```
      J = 3
C
C In binary,
C J = 00000000 00000000 00000000 00000011
C
      I = ISHFT(J,2)
C
C The value returned to I is 12; in binary,
C I = 00000000 00000000 00000000 00001100.
C
      J = 3
C
C In binary,
C J = 00000000 00000000 00000000 00000011
C
      I = ISHFT(J,-1)
C
C The value returned to I is 1; in binary,
C I = 00000000 00000000 00000000 00000001.
C
```

October 1983

BTEST

Purpose: To test whether a specific bit in a fullword integer is set to one.

Location: *LIBRARY

Calling Sequence:

```
LOGICAL a,BTEST
a = BTEST(j,k)
```

Parameters:

- j is the fullword (INTEGER*4) integer to be tested.
- k is an integer specifying the number of the bit in j to be tested.
- a is a logical variable which is set to TRUE if bit k in j is equal to one, and FALSE, otherwise.

Description: TRUE is returned if bit k in integer j is equal to one; if it is not, FALSE is returned. Bits are numbered from right to left, from 0 to 31. If k is less than zero or greater than 31, an error message is printed and execution is terminated.

In languages other than FORTRAN, the returned value is an integer, with 1 for TRUE and 0 for FALSE. (FORTRAN programs may also declare BTEST as an integer function, although the standards specify the function type to be logical.)

Example:

```
      J = 10
C
C In binary,
C J = 00000000 00000000 00000000 00001010
C
      LOGICAL A,B,BTEST
      A = BTEST(J,3)
      B = BTEST(J,2)
C
C The value returned to A is TRUE; the value
C returned to B is FALSE.
C
```

IBSET

Purpose: To set a specific bit of a fullword integer to one.

Location: *LIBRARY

Calling Sequence:

```
i = IBSET(j,k)
```

Parameters:

- j is a fullword (INTEGER*4) integer.
- k is a fullword integer specifying the number of the bit in j to be set.
- i is a fullword integer containing j with the k'th bit set to one.

Description: Bit number k of integer j is set to one. Bits are numbered from right to left, from 0 to 31.

If k is less than zero or greater than 31, an error message is printed and execution is terminated.

Example:

```
      J = 8
C
C In binary,
C J = 00000000 00000000 00000000 00001000
C
      I = IBSET(J,2)
C
C The value returned to I is 12; in binary,
C I = 00000000 00000000 00000000 00001100.
C
```

October 1983

IBCLR

Purpose: To clear (set to zero) a particular bit in a fullword integer.

Location: *LIBRARY

Calling Sequence:

```
i = IBCLR(j,k)
```

Parameters:

- j is a fullword (INTEGER*4) integer.
- k is a fullword integer specifying the number of the bit in j to be cleared.
- i is a fullword integer containing the value of j with bit number k cleared.

Description: Bit number k in integer j is set to zero. Bits are numbered from right to left, from 0 to 31.

If k is less than zero or greater than 31, an error message is printed and execution is terminated.

Example:

```
      J = 12
C
C In binary,
C J = 00000000 00000000 00000000 00001100
C
      I = IBCLR(J,3)
C
C The value returned to I is 4; in binary,
C I = 00000000 00000000 00000000 00000100.
C
```

ISHFTC

Purpose: To shift all or part of a fullword integer left or right in a circular fashion.

Location: *LIBRARY

Calling Sequence:

```
i = ISHFTC(j,k,l)
```

Parameters:

- j is a fullword (INTEGER*4) integer whose bits are to be shifted.
- k is a fullword integer indicating the number of positions to be shifted.
- l is a fullword integer indicating the number of bits to be shifted.
- i is a fullword integer containing the value of j with the rightmost l bits shifted circularly k positions.

Description: The rightmost l bits in j are shifted left (if k>0) or right (if k<0) by k positions.

The shift is circular--a bit shifted out of the left side (leftward shift) or the right side (rightward shift) is moved directly to the opposite side of the string of bits being shifted.

If the number of bits to be shifted l is less than one or greater than 32, or if the number of positions to be shifted is greater than the number of bits (i.e., k > l), an error message is printed on SERCOM and execution is terminated.

Example:

```
      J = 6
C
C In binary,
C J = 00000000 00000000 00000000 00000110
C
      I = ISHFTC(J,2,4)
C
C The value returned to I is 9; in binary,
C I = 00000000 00000000 00000000 00001001.
C
```

October 1983

IBITS

Purpose: To extract a string of bits from a fullword integer.

Location: *LIBRARY

Calling Sequence:

```
i = IBITS(j,k,l)
```

Parameters:

- j is a fullword (INTEGER*4) integer from which bits are to be extracted.
- k is a fullword integer indicating the first bit of j to be extracted. The bits are numbered right to left from 0 to 31.
- l is a fullword integer indicating the number of bits to be extracted.
- i is a fullword integer containing the right-justified bit string extracted from j.

Description: Bits k through k+l-1 in j are right-justified and returned in i.

If k is greater than 31, l is less than or equal to zero, or k+l is greater than 32, an error message is printed and execution is terminated.

Example:

```
      J = 10
C
C In binary,
C J = 00000000 00000000 00000000 00001010
C
      I = IBITS(J,1,3)
C
C The value returned to I is 5; in binary,
C I = 00000000 00000000 00000000 000000101
C
```

MVBITS

Purpose: To move a string of bits from one binary integer to another.

Location: *LIBRARY

Calling Sequence:

```
CALL MVBITS(m,i,len,n,j)
```

Parameters:

- m — is the fullword (INTEGER*4) integer from which the string is to be taken.
- i — is a fullword integer containing the number of the first bit of m in the string. Bits are numbered right to left from 0 to 31.
- len — is a fullword containing the length of the string.
- n — is a fullword integer into which the bit string is inserted.
- j — is a fullword integer containing the number of the first bit of the area in n into which the string is to be moved.

Description: Bits i through i+len-1 of m are moved to positions j through j+len-1 of n. The bits of n not in the range j through j+len-1 remain unchanged.

If the string length len is less than one or greater than 32, or if either bit offset (i or j) is less than zero or greater than 31, or if either bit offset plus the length (i+len or j+len) is greater than 32, an error message is printed on SERCOM and execution is terminated.

Example:

```
      M = 53
      N = 74
C
C In binary,
C M = 00000000 00000000 00000000 00110101
C N = 00000000 00000000 00000000 01001010
C
      CALL MVBITS(M,1,4,N,4)
C
C
C The value returned to N is 170; in binary,
C N = 00000000 00000000 00000000 10101010.
C
```

October 1983

DATE

Purpose: To return the current date.

Location: *LIBRARY

Calling Sequence:

```
INTEGER i(3)
CALL DATE(i)
```

Parameter:

i is the start of a three-element integer array in which the current date is returned.

Description: Returns the current date in integer array i, with the year (since AD 0) in the first element, the month (1-12) in the second element, and the day (1-31) in the third.

Example: If the subroutine were called on May 1, 1983, on return, i(1) would contain 1983, i(2) would contain 5, and i(3) would contain 1.

ANSITM

Purpose: to return the current time.

Location: *LIBRARY

Calling Sequence:

```
INTEGER i(3)
CALL ANSITM(i)
```

Parameter:

i is a three-element integer array in which the current time is returned.

Description: The current time is returned in the first three elements of the specified array, with the hour (0-23) in the first element, the current minute (0-59) in the second element, and the current second (0-59) in the third.

Example: If the call were made at 10:42:30 PM, i(1) would contain 22, i(2) would contain 42, and i(3) would contain 30.

Note: In the ANSI standard, the name of this subroutine is TIME; the name has been changed here because there is a different function under MTS named TIME.

Programmers using the ANSITM subroutines have two alternatives: first, they can change their source programs to call ANSITM rather than TIME; or they can use the object file editor (*OBJUTIL) to change calls to TIME into calls to ANSITM. For example,

```
$RUN *OBJUTIL 0=objectfile
RENAME TIME ANSITM
STOP
```

where "objectfile" is the file containing the compiled program. This sequence will change the subroutine name from TIME to ANSITM for every CALL TIME statement in the program.

October 1983

ANSI STANDARD FILE CONTROL SUBROUTINES

This set of subroutines contains procedures for file control as described in ANSI/ISA-S61.2, Industrial Computer System FORTRAN Procedures for File Access and the Control of File Contention.

These subroutines are intended to allow FORTRAN programs written for other systems, which provide subroutines implementing the same standards, to be run under MTS with little or no modification, and to facilitate the development under MTS of FORTRAN programs intended for use on such systems.

The following subroutines are available:

Subroutine	Function
CFILW	Create a file
DFILW	Destroy a file
OPENW	Open a file
CLOSEW	Close a file
MODAPW	Modify access privileges for an open file
RDRW	Read a record from a file
WRTRW	Write a record to a file

Note: These subroutines only provide for direct access to files.

The following list describes all extensions to and incompatibilities with the standard.

(1) The standards make no specific mention of the handling of calls with invalid parameters. In this implementation, the return code for each subroutine is set to indicate the type of error detected.
(2) File names are not covered by the standards, but left dependant on the processor. These subroutines expect file names to be standard MTS file names, terminated by a blank space. (This can be effected in full accord with the standard by using integer arrays initialized to contain the file names.)
(3) The standards permit concurrently executing programs to write to the same file and allow one program to read a file while a concurrent program is writing to it; under MTS such access is not possible. Therefore, a program requesting write access to a file will receive it only if no other program is accessing the file in any way.
(4) The standards specify that an open file is attached to a particular unit, and use the unit number to identify the file. These subroutines make use of the unit numbers as specified, but do not actually associate the units with the MTS logical I/O

units. Thus, it would be possible to have a file open under the ANSI file subroutines, attached to unit 5, and to have a different file open and attached to MTS unit 5. Note also that MTS logical I/O units run from 0 to 99, while the ANSI subroutines allow the unit number to be any integer.

(5) A file that is open may be destroyed. This might cause an error return if I/O is subsequently attempted to the file.

October 1983

CFILW

Purpose: To create a file.

Location: *LIBRARY

Calling Sequence:

CALL CFILW(j,n1,n2,m)

Parameters:

- j is an integer array containing the name of the file followed by one blank space. The file name must conform to standard MTS conventions.
- n1 is an integer specifying the number of bytes per record in the file.
- n2 is an integer specifying the number of records in the file.
- m is an integer variable to receive a return code.

Return codes:

1	File successfully created.
2	Bytes per record value zero or less.
3	Number of records less than zero.
104	File already exists.
128	Space allocated to this CCID exceeded.

(Errors 108,112,116,120,124 should not occur). Errors 104-128 are return codes from the CREATE subroutine, plus 100 (see MTS Volume 3, System Subroutine Descriptions, for details).

Description: A file with the specified file name is created, if possible. The record size and file size are used to compute the approximate number of disk pages required to hold the file, and the file is created at that size.

Because the formula used is only an approximation of the file space required, the file might still require expansion when records are written to it (see Appendix C to the section "Files and Devices" in MTS Volume 1, The Michigan Terminal System, for a description of the formula used). It should also be noted that the values specified in the CFILW call are used only in computing the initial size of the file; they do not determine actual record size or the maximum size of the file. (This leniency is not part of the standard, but a feature of the MTS file system; under some systems, all charac-

teristics for a file must be specified when it is created.)

Example:

```
INTEGER FILNAM(3)/'FILE','1 '/
CALL CFILW(FILNAM,1,1,m)
```

The above sequence creates a file named FILE1.

October 1983

DFILW

Purpose: To destroy a file.

Location: *LIBRARY

Calling Sequence:

CALL DFILW(i,m)

Parameters:

- i is an integer array containing the name of a file followed by a blank space. Standard MTS conventions for file names should be followed.
- m is an integer variable to receive the return code.

Return Codes:

- 1 File successfully destroyed.
- 4-28: Return codes from the DESTROY subroutine (see MTS Volume 3, System Subroutine Descriptions, for details).

Example:
```
INTEGER FILNAM(3)/'FILE','22 '/
```

Description: The specified file is destroyed.
```
CALL DFILW(FILNAM,M)
```

The above sequence destroys the file named FILE22.

OPENW

Purpose: To open a file for input/output.

Location: *LIBRARY

Calling Sequences:

CALL OPENW(i,j,k,m)

Parameters:

- i is an integer array containing an MTS file name followed by a blank space. Standard MTS conventions for file names should be followed, and the file name should not include line ranges or I/O modifiers.
- j is the unit number to be associated with the file. This integer number identifies the file for all subsequent processing.
- k is the type of access to the file required (see below).
- m is an integer variable to receive the return code.

Return Codes:

1	File successfully opened.
2	Invalid access requested.
3	Access (under MTS) insufficient to open file SHARED or EXCLUSIVE ALL.
4	Unit already in use.
104	File does not exist
108	Hardware error or software inconsistency.
112	Access not allowed.
208	File is busy.
212	File not operational.
304	File does not exist.
308	Hardware error or software inconsistency.
312	Access appropriate to lock not allowed.
316	Locking would cause deadlock.
320	File locked by another task.

Errors 104 to 112 are return codes from the CHKFILE subroutine, plus 100. Errors 208 and 212 are return codes from the GETFD subroutine, plus 200. Errors 304 to 320 are return codes from the LOCK subroutine, plus 300 (see MTS Volume 3, System Subroutine Descriptions, for details).

October 1983

Description: The specified file is opened with the desired access, if possible. The following are valid access modes:

1	READ ONLY:	The calling program can read the file; other concurrent programs may read or write.
2	SHARED:	The calling program can read and write; other concurrent programs may read or write.
3	PROTECTED READ:	The calling program can only read; other concurrent programs may only read.
4	EXCLUSIVE ALL:	The calling program can read and write; other concurrent programs may not access the file.

MTS does not permit two tasks to write to the same file concurrently, or allow one task to read a file while another task is writing to that file. Therefore, a request for READ ONLY access will be treated as a request for PROTECTED READ access, and a request for SHARED access will be treated as a request for EXCLUSIVE ALL access.

Example:
```
INTEGER FILNAM(3)/'FILE','333 '/
INTEGER UNIT/22/,ACCESS/3/,M
CALL OPENW(FILNAM,UNIT,ACCESS,M)
```

The above sequence opens the file named FILE333 for PROTECTED READ access on unit number 22.

MODAPW

Purpose: To modify the calling program's access privileges to an open file.

Location: *LIBRARY

Calling Sequence:

CALL MODAPW(i,j,m)

Parameters:

- i is the unit number of the file to be processed.
- j is the new access privilege requested for the file.
- m is an integer variable to receive the return code.

Return Codes:

1	Access successfully changed.
2	Invalid access mode specified.
3	Unit not associated with a file.
104	File does not exist (should not occur).
108	Hardware error or software inconsistency.
112	Access appropriate to locking request not allowed.
116	Locking as requested would result in a deadlock.
120	File locked by another task.

Errors 104 through 120 are return codes from the LOCK subroutine, plus 100 (see MTS Volume 3, System Subroutine Descriptions, for details).

Description: The program's access to the specified file is changed to the specified access, if possible.

Example:

```
INTEGER UNIT/23/,ACCESS/4/
CALL MODAPW(UNIT,ACCESS,M)
```

The above sequence would result in the access to the file attached as unit 23 being changed to EXCLUSIVE ALL (if possible).

October 1983

CLOSEW

Purpose: To close a file.

Location: *LIBRARY

Calling Sequence:

CALL CLOSEW(i,m)

Parameters:

- i is the unit number attached to the file to be closed.
- m is an integer variable to receive the return code.

Return Codes:

- 1 File closed.
- 2 Unit number not attached to a file.

Description: The specified file is released. It may not be accessed by other I/O subroutines unless it is reopened by a call to OPENW.

Example:

```
INTEGER UNIT/26/,M
CALL CLOSEW(UNIT,M)
```

The above sequence closes the file attached to unit number 26.

RDRW

Purpose: To read a record from a file.

Location: *LIBRARY

Calling Sequence:

CALL RDRW(i,j,k,l,m)

Parameters:

- i is the number of the unit attached to the file.
- j is the number of the record to be read. This number must be a positive integer.
- k is the start of the area into which the record is to be read.
- l is the number of bytes to be read. If the actual record is longer than this, it is truncated. If the record is shorter, it is padded with blanks (X'00').
- m is an integer variable to receive the return code.

Return Codes:

1	Record read successfully
2	Invalid record number specified.
3	Invalid length specified.
4	Unit not attached to a file.
104	Requested record not in file.

Errors 104 and greater are return codes from the READ subroutine plus 100. A return code from READ greater than 4 will cause an error to be printed and execution to be terminated (see MTS Volume 3, System Subroutine Descriptions, for details).

Description: The specified line is read from the file into the designated area. If necessary, the record may be padded with nulls (X'00' bytes) or truncated.

Example:

```
INTEGER UNIT/33/,RECNUM/6/,INBUF(4)
INTEGER LENGTH/12/,M
CALL RDRW(UNIT,RECNUM,INBUF,LENGTH,M)
```

The above sequence will read twelve bytes (3 full-words) into INBUF(1),INBUF(2), and INBUF(3) from record 6 of the file attached to unit 33.

October 1983

WRTRW

Purpose: To write a record to a file.

Location: *LIBRARY

Calling Sequence:

CALL WRTRW(i,j,k,l,m)

Parameters:

- i is the number of the unit attached to the file.
- j is the number of the record to be written to. This must be a positive integer.
- k is the start of the area from which the record is to be written.
- l is the number of bytes to be written.
- m is an integer variable to receive the return code.

Return Codes:

1. Record successfully written.
2. Invalid record number.
3. Invalid length.
4. No file attached to specified unit.
5. Access to file is READ ONLY or PROTECTED READ.

Description: The specified record in the specified file is written from the indicated location.

Example:

```
INTEGER UNIT/40/,RECNUM/12/
INTEGER OUTBUF(6)/1,2,3,4,5,6/,LEN/24/,M
CALL WRTRW(UNIT,RECNUM,OUTBUF,LEN,M)
```

The above sequence will write 24 bytes (6 fullwords to record number 12 of the file attached to unit number 40 from the integer variable OUTBUF.

October 1983

October 1983

MISCELLANEOUS FORTRAN SUBROUTINES

The following subroutine descriptions are taken from MTS Volume 3, System Subroutine Descriptions. These are subroutines which may be of use to FORTRAN users.

ADROF

Purpose: To return the address of a FORTRAN variable.

Location: *LIBRARY

Alt. Entry: IADROF

Calling Sequences:

FORTRAN: x = ADROF(var)

Parameters:

var is the location of the variable name whose address is to be returned. If the variable name is a character string which is intended to be used as an FDname, it should be terminated with a trailing blank.

Values Returned:

x will contain the address of the variable.

Note: In FORTRAN, ADROF should be declared as an INTEGER*4 function. ADROF is intended for use with RCALL to compute addresses as necessary in calling R-type subroutines (see the RCALL subroutine description in this volume).

Example: FORTRAN:

```
INTEGER*4 RESULT,ADROF
      .
      .
RESULT = ADROF('FDname ')
```

This example returns the address of the character string "FDname" in the variable RESULT.

October 1983

ATNTRP

Purpose: To allow a FORTRAN program to be notified of the occurrence of an attention interrupt.

Location: *LIBRARY

Calling Sequence:

FORTRAN: CALL ATNTRP(flag)

Parameter:

flag is a LOGICAL*4 variable which will be set to .TRUE. when an attention interrupt occurs.

Return Codes:

None.

Description: A call to the ATNTRP subroutine will set the value of flag to .FALSE. and will enable the attention interrupt trap. When an attention interrupt occurs, flag will be set to .TRUE., the trap will be disabled, and execution of the interrupted program will be resumed at the point of the interrupt. It is the responsibility of the FORTRAN program to detect a change in the value of flag and to act accordingly.

One call to ATNTRP allows only one attention interrupt to be intercepted. To intercept another attention interrupt, ATNTRP must be called again.

Example: FORTRAN:

```
      LOGICAL*4 FLAG
      CALL ATNTRP(FLAG)
        .
        .
   10 IF(FLAG) GO TO 20
        .
        .
      GO TO 10
   20 CONTINUE
        .
        .
```

This example calls ATNTRP to enable the intercept of one attention interrupt. Periodically, the program checks the value of FLAG to determine if an interrupt has occurred; if an interrupt has occurred, a branch is made to statement label 20.

CHKPAR

Subroutine Description

Purpose: To check the number and data types of parameters passed to a subroutine.

Location: *LIBRARY

Calling Sequences:

FORTRAN: CALL CHKPAR(icode,'string ',&rc4)

Parameters:

icode is a switch indicating the action to be taken if an error is found by CHKPAR. The legal switch values are:

0 A traceback of the subroutine calls is produced and then execution is suspended. Execution may be resumed by the $RESTART command.
1 A traceback of the subroutine calls is produced and then execution is resumed.
2 Execution is continued with an error message but without a traceback.
3 Execution is continued without an error message or a traceback.

In all cases, a return code 4 (RETURN 1) is produced if an error is detected.

string is a string of characters of the form I (integer), R (real), and X (other) which corresponds in data type to the dummy variables in the calling sequence of the subroutine being checked. CHKPAR checks only REAL*4 and REAL*8 variables, and INTEGER*4 variables of magnitude less than 1048575. All other variables must be indicated by an X and are ignored. The string must be enclosed in primes and terminated by a blank.

The letter O may be included in the string to indicate that the remaining parameters are optional. The letter S may be included to stop the checking of parameters before the end of the parameter list is encountered. The S option is useful if the caller is not

required to set the variable length bit (the high-order bit in the last parameter address).

CHKPAR will not differentiate between REAL*4 and REAL*8 variables.

rc4 (optional) is the number of a FORTRAN statement to transfer to if the number of parameters or their data types are not correct. If omitted, control will return to the statement following the call to CHKPAR.

Note: Standard OS Type-I(S) calling conventions must be used in all subroutine calls. See the section "Calling Conventions" in MTS Volume 3, System Subroutine Descriptions.

Description: CHKPAR tests the data types of the arguments in the subroutine from which CHKPAR was called against the data types specified in the string parameter. A value of zero is legal regardless of data type. If the value is nonzero, the absolute value of the variable is taken and the high-order byte is tested for zero. If this byte is nonzero, the corresponding data type must be R. If this byte is zero, the next 4 bits (20-23) must be zero for integer variables and nonzero for real variables.

CHKPAR must be called from the subroutine whose parameter list is being checked.

Examples: FORTRAN:

```
      X=10.
      Y=20.
      CALL SUBR(X,Y,Z)
      STOP
      END

      SUBROUTINE SUBR(I,Y,Z)
      CALL CHKPAR(1,'IRX ',&10)
      Z=FLOAT(I)+Y
      RETURN
 10   WRITE(6,100)
100   FORMAT('0ERROR IN CALL TO SUBR')
      STOP
      END
```

In the above example, X is incorrect in the call to SUBR. The following type of message is subsequently printed:

```
Error in argument number n in call to subroutine SUBR.
Type should be (integer/real) is (real/integer).
Integer value is "xxxx", real "xxxx", hex "xxxx",
character "xxxx".
```

CHKPAR then produces a traceback and transfers control to statement number 10. The third parameter Z in the above example is not checked by CHKPAR because it is returned by the subroutine SUBR and therefore is not initialized when CHKPAR is called.

```
FORTRAN:        I=10.
                Y=20.
                CALL SUBR(I,Y)
                STOP
                END

                SUBROUTINE SUBR(I,Y,Z)
                CALL CHKPAR(0,'IRX ',&10)
                Z=FLOAT(I)+Y
                RETURN
           10   WRITE(6,100)
          100   FORMAT('0ERROR IN CALL TO SUBR')
                STOP
                END
```

In the above example, the following message is printed:

   Number of arguments wrong in call to SUBR.

CHKPAR then produces a traceback and suspends execution. The user may resume execution via the $RESTART command.

October 1983

DUMP, PDUMP

Purpose: To print the values of specified memory regions in a FORTRAN program.

Location: *LIBRARY

Calling Sequence:

FORTRAN: CALL DUMP(a1,b1,f1,...,an,bn,fn)

CALL PDUMP(a1,b1,f1,...,an,bn,fn)

Parameters:

ai is a variable in the FORTRAN program specifying one end of the "i"th region to be printed.

bi is a variable in the FORTRAN program specifying the other end of the "i"th region to be printed.

fi indicates the format in which each data item between ai and bi is to be printed. fi is a fullword integer and may be one of the following values:

0 - hexadecimal
1 - LOGICAL*1
2 - LOGICAL*4
3 - INTEGER*2
4 - INTEGER*4
5 - REAL*4
6 - REAL*8
7 - COMPLEX*8
8 - COMPLEX*16
9 - literal

Description: The DUMP and PDUMP subroutines print the values of the data items in the memory regions delimited by the ai and bi parameters. As many triples of parameters, ai, bi, and fi, may be given as desired. There is no order implied by the ai and bi parameters--either may mark the beginning or end of a region to be dumped. All output is printed on the logical I/O unit SERCOM.

The relative locations of the variables in a FORTRAN program may be obtained from the map produced by the MAP option to the FORTRAN compiler.

The only difference between DUMP and PDUMP is that DUMP terminates execution of the calling program by calling the system subroutine SYSTEM while PDUMP returns to the calling program.

Example: FORTRAN: CALL DUMP(A(1),A(100),5,A(1),A(100),0)

The above example prints the values of the first 100 elements of the array A in both REAL*4 and hexadecimal format.

October 1983

GDINF

Purpose: To allow a FORTRAN program to obtain information returned from the subroutine GDINFO.

Location: *LIBRARY

Calling Sequence:

FORTRAN: CALL GDINF(unit,region,&rc4)

Parameters:

- unit — is the location of either
  (a) a FDUB-pointer (as returned by GETFD),
  (b) an 8-character logical I/O unit name left-justified with trailing blanks (e.g., SCARDS, SPRINT, 0 through 99, etc.), or
  (c) an integer logical I/O unit number (0-99).
- region — is a 44-byte array (11 fullwords) in which the information is returned.
- rc4 — (optional) is the statement label to transfer to if a nonzero return code occurs.

Return Codes:

0 Successful return.
4 Error. See the GDINFO subroutine description for the possible error conditions.
8 Hardware or software inconsistency.

Description: This subroutine calls the GDINFO subroutine and places the returned information in region which is provided by the FORTRAN calling program. See the description of the GDINFO subroutine in MTS Volume 3, System Subroutine Descriptions, for a description of this information. Note that only the first eleven words of GDINFO information is returned.

Example: FORTRAN:

```
      INTEGER*4 REG(11)
       ...
      CALL GDINF('SPUNCH  ',REG,&99)
       ...
   99 WRITE(6,199)
  199 FORMAT(' SPUNCH IS NOT ASSIGNED')
```

This example calls GDINF to obtain information about the file or device attached to SPUNCH.

October 1983

NPAR

Subroutine Description

Purpose: To count the number of parameters passed to a subroutine.

Location: *LIBRARY

Calling Sequences:

FORTRAN: i = NPAR(n)

Parameters:

n is the number of subroutine or function calls to be counted. That is, a value of 1 will return the number of parameters passed to the subroutine in which NPAR is called. A value of 2 would return the number of parameters passed to the subroutine that called the subroutine that called NPAR. For most uses, n will be 1. An error message is generated if n exceeds the nesting level of the subroutine calling NPAR.

Multiple return statement numbers are not counted as parameters by NPAR.

i is number of parameters passed.

Notes: Standard OS Type-I(S) calling conventions must be used in all subroutine calls. See the section "Calling Conventions" in MTS Volume 3, System Subroutine Descriptions.

If the subroutine calling NPAR has more parameters in its parameter list than are provided by its caller, then the excess parameters must be enclosed in slashes. Otherwise, a program interrupt may occur during the entry prolog code to the subroutine.

Example: FORTRAN:

```
CALL SUBR(X)
STOP
END

SUBROUTINE SUBR(/X/,/Y/,/Z/)
I = NPAR(1)
IF (I .GE. 4) GO TO 10
IF (I .EQ. 3) GO TO 300
IF (I .EQ. 2) GO TO 200
```

```
      IF (I .EQ. 1) GO TO 100
10    WRITE(6,11)
11    FORMAT('ERROR')
        ...
100     ...
200     ...
300     ...
        ...
      RETURN
      END
```

In the above example, NPAR counts the number of parameters passed to SUBR and sets up a branch accordingly. In this case, one parameter was passed.

RCALL

Purpose: To call R-type subroutines (such as GETFD) from FORTRAN.

Location: *LIBRARY

Calling Sequences:

FORTRAN: CALL RCALL(a,m,ir(1),...,ir(m),n,rr(1),...,
rr(n),&rc4,...)

Parameters:

- a is the address of the R-type subroutine which is to be called. This should be declared EXTERNAL.
- m is the fullword integer number of general registers starting with GR0 to be set up prior to calling the R-type subroutine. m may range between 0 and 13, inclusive.
- ir(1),...,ir(m) are the values to be placed in GR0 through GR(m-1), respectively. These parameters must be fullword-aligned and four bytes in length.
- n is the fullword integer number of general registers starting with GR0 to be stored after calling the R-type subroutine. n may range between 0 and 13, inclusive.
- rr(1),...,rr(n) are the n variables into which the contents of GR0 through GR(n-1) will be stored after calling the R-type subroutine. These parameters must be fullword-aligned and four bytes in length.
- rc4,... is the statement label to transfer to upon receiving a nonzero return code from the subroutine called via RCALL.

Return Codes:

The return code from RCALL is identical to the return code returned by the R-type subroutine. The contents of the general registers have been returned after the R-type subroutine call as specified by the parameters.

Description: The general registers starting with 0 are set up as specified by the parameter list. The second parameter specifies the number of registers to be set up, and the parameters following specify the values to be placed into the registers. The R-type subroutine is called, and when it returns, the general registers starting with 0 are

stored as specified by the parameter list. The return code is as returned by the R-type subroutine.

Many R-type subroutines require that addresses be placed in registers before calling them. These addresses can be computed by using the subroutine ADROF. See the ADROF subroutine description in this volume.

If the subroutine also requires an S-type parameter list, the address of the parameter list must be placed in GR1. This may be done by using the ADROF subroutine, where the argument to ADROF is a scalar variable for a single-element parameter list or an array for a multiple-element parameter list.

Example:

```
FORTRAN:   EXTERNAL GETFD
           INTEGER*4 ADROF,FDUB
           CALL RCALL(GETFD,2,0,ADROF('name '),1,FDUB,&9)
```

This example calls GETFD with GR0 containing a zero and GR1 containing the address of the character string "name". GETFD returns the FDUB-pointer in GR0, and this is stored in the variable FDUB. A return code of four from GETFD will cause control to be transferred to statement 9 of the FORTRAN program.

```
FORTRAN:   EXTERNAL CHKFIL
           INTEGER*4 ADROF,X
           DATA MASK/Z00000001/
           PAR = ADROF('2AGA:DATAFILE ')
           CALL RCALL(CHKFIL,2,0,ADROF(PAR),1,X,&100)
           X = LAND(X,MASK)
           IF(X.EQ.1) GO TO 10
```

This example illustrates a call to the subroutine CHKFIL, which uses both an S-type calling sequence parameter list and a R-type return of a value. In this case, the first parameter to CHKFIL is the location of the name of a file.

REWIND

Subroutine Description

Purpose: To rewind a logical I/O unit in FORTRAN.

Location: *LIBRARY

Calling Sequences:

FORTRAN: CALL REWIND(unit)

Parameters:

unit is the location of a fullword integer corresponding to the logical I/O unit number to be rewound. These are 0 through 99.

Description: If the logical I/O unit number specified by unit is attached to a magnetic tape, it is rewound. If it is attached to a line file, it is reset so that the next sequential reference to it will read or write the line specified by the beginning line number given when the file was attached. If it is attached to a sequential file or a floppy disk, it is reset so that the next reference to it will read or write from the beginning of the file. In all other cases, an error comment is produced on the logical I/O unit SERCOM, and the subroutine ERROR is called.

The REWIND subroutine generates a call to the REWIND# subroutine.

Example: FORTRAN: CALL REWIND(1)

The file or device attached to logical I/O unit 1 is rewound.

SIOERR

Purpose: To allow FORTRAN users to regain control when I/O transmission errors that would otherwise be fatal (such as tape I/O errors or exceeding the size of a file) occur during execution.

This subroutine is obsolete. The @ERRRTN I/O modifier, the FORTRAN ERR exit feature, or the error recovery features of the FTNCMD subroutine should be used instead.

Location: *LIBRARY

Calling Sequence:

FORTRAN:
```
EXTERNAL subr
CALL SIOERR(subr)
```

Parameters:

subr is the subroutine to transfer to when an I/O error occurs, or zero, in which case the error exit is disabled.

Description: A call on the subroutine SIOERR sets up an I/O transmission error exit for one error only. When an error occurs and the exit is taken, the intercept is cleared so that another call to SIOERR is necessary to intercept the next I/O transmission error.

| If the logical I/O unit specified by unit is part of an
| explicit concatenation, only the currently active member
| is rewound.

If the subroutine subr returns, a return is made to the user's program from the I/O routine with the return code indicating the type of error that occurred. The return code depends upon the type of device in use when the error occurred. See the section "I/O Subroutine Return Codes" in MTS Volume 1, The Michigan Terminal System.

Note: SETIOERR is for assembly language (see the description of the subroutine SETIOERR in MTS Volume 3) and SIOERR is for FORTRAN users. There is a difference in the level of indirection between the two subroutines; therefore, SIOERR should not be used by assembly language users.

Many I/O error conditions are detected by the FORTRAN I/O library before they actually occur,

thus allowing the FORTRAN monitor to take corrective action. In these cases, an error exit enabled by a call to SIOERR will not be taken since the FORTRAN monitor will take control before the erroneous operation is attempted. For further details, see the section "FORTRAN I/O Library" in this volume.

Example: FORTRAN:

```
EXTERNAL SWITCH
COMMON ISW
     .
     .
ISW=0
CALL SIOERR(SWITCH)
WRITE (8,105) FILEOUT
IF(ISW.EQ.1) GO TO 10
CALL SIOERR(0)
     .
     .
SUBROUTINE SWITCH
COMMON ISW
ISW=1
RETURN
END
```

In this example, SIOERR is called to enable an exit if an I/O error occurs during the processing of the WRITE statement. If an error does occur, the subroutine SWITCH will be called which sets the variable ISW to 1 and returns. The calling program tests the value of ISW and branches to statement 10 if appropriate. SIOERR is called again to disable the exit.

October 1983

*PROFORT: THE FORTRAN EXECUTION PROFILER

The Profiler is a tool for analyzing the performance of FORTRAN programs. The Profiler first runs a FORTRAN program, and then produces a profile of the program comprised of a source-code listing and a flow graph showing how many times each source statement was executed during the run and how many times each path of the program was traversed.

Every FORTRAN program can be represented as a flow graph whose nodes are the program's executable statements and whose edges are the paths between statements.

A simple FORTRAN program is presented below. The numbers in the first column at the left are the MTS line numbers of the source file. The numbers in the second column are the familiar ISNs (Internal Statement Numbers), like those provided on source listings by the FORTRAN G Compiler.

Figure 1 shows the flow graph of this program with ISNs representing the actual statements. Note that logical IFs are shown as two statements, the first being the evaluation of the logical expression, the second being the statement to execute if the logical expression is true. This separation makes sense because each half of a logical IF has its own predecessor and successor paths and its own execution history. The convention adopted is to assign the ISNs N and N' to the two halves of a logical IF.

Figure 2 shows the source listing of the program integrated with the flow graph of Figure 1. This is the way the Profiler displays program flow graphs. An asterisk before an ISN indicates that this path is the result of normal sequencing from one statement to the next. A pound sign indicates that this is a DO-loop return path (from the last to the first executable statement in a DO range).

Figure 3 shows an execution profile of the same program. This is the source listing with flow graph of Figure 2 supplemented by execution frequencies. The column labeled COUNT shows the number of times each statement was executed during the run. The column labeled PRED:COUNT shows the number of times each predecessor path was traversed and the column labeled SUCC:COUNT shows the same for successor paths.

In this particular execution SUBROUTINE OCCUR was called twice. On the first call LEN was 10 and 2 occurrences of VALUE were found. On the second call LEN was 20 and 3 occurrences of VALUE were found. The profile corroborates these facts.

```
801        C This routine searches an integer 'array' of
802        C length 'len' for occurrences of 'value' and
803        C returns the no. of such occurrences in 'num'.
804        C
805    1         SUBROUTINE OCCUR (ARRAY,LEN,VALUE,NUM)
806    2         INTEGER ARRAY(LEN),VALUE
807    3         NUM=0
808    4         IF (LEN.LE.0) RETURN
809    5         DO 100 I=1,LEN
810    6         IF (VALUE.EQ.ARRAY(I)) NUM=NUM+1
811    7       100 CONTINUE
812    8         RETURN
813    9         END
```

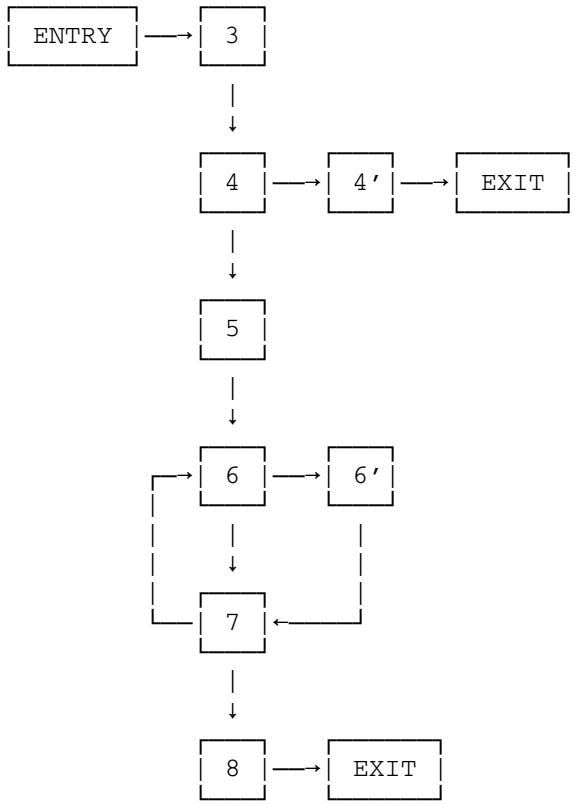


Figure 1: Flow Graph

LNR		ISN	PRED	SUCC
801	C This routine searches an integer 'array' of			
802	C length 'len' for occurrences of 'value' and			
803	C returns the no. of such occurrences in 'num'			
804	C			
805	SUBROUTINE OCCUR (ARRAY,LEN,VALUE,NUM)	1		
806	INTEGER ARRAY(LEN),VALUE	2		
807	NUM=0	3	ENTRY	*4
808	IF (LEN.LE.0)	4	*3	4'
				*5
808	+ RETURN	4'	4	EXIT
809	DO 100 I=1,LEN	5	*4	*6
810	IF (VALUE.EQ.ARRAY(I))	6	*5	6'
			#7	*7
810	+ NUM=NUM+1	6'	6	*7
811	100 CONTINUE	7	*6	#6
			*6'	*8
812	RETURN	8	*7	EXIT
813	END	9		

Figure 2: Profiler Flow Graph

LNR		ISN	COUNT	PRED : COUNT	SUCC : COUNT
801	C This routine searches an integer 'array' of				
802	C length 'len' for occurrences of 'value' and				
803	C returns the no. of such occurrences in 'num'				
804	C				
805	SUBROUTINE OCCUR (ARRAY,LEN,VALUE,NUM)	1			
806	INTEGER ARRAY(LEN),VALUE	2			
807	NUM=0	3	2	ENTRY : 2	*4 : 2
808	IF (LEN.LE.0)	4	2	*3 : 2	4' : 0
					*5 : 2
808	+ RETURN	4'	0	4 : 0	EXIT : 0
809	DO 100 I=1,LEN	5	2	*4 : 2	*6 : 2
810	IF (VALUE.EQ.ARRAY(I))	6	30	*5 : 2	6' : 5
				#7 : 28	*7 : 25
810	+ NUM=NUM+1	6'	5	6 : 5	*7 : 5
811	100 CONTINUE	7	30	*6 : 25	#6 : 28
				*6' : 5	*8 : 2
812	RETURN	8	2	*7 : 2	EXIT : 2
813	END	9			

Figure 3: Execution Profile

The Value of a Profiler

One strategy for testing programs is to feed the program a sufficient variety of test data (not necessarily in the same test run) to cause every possible path to be traversed at least once. This does not guarantee an error-free program, but at least it leaves no code entirely untested. A profiler is a useful tool for applying this strategy. One looks for zero counts and creates additional test data to eliminate them.

A profiler is also useful for exposing errors, by virtue of its power to reveal erroneous loop counts, excursions into inappropriate portions of code, and other unexpected behavior.

The information provided by a profiler forms a good basis for analyzing algorithms and their implementations in code. The most frequently executed portions of a program are often those where the most time is spent. Consequently, high execution frequencies are a good indicator of program bottlenecks.

Donald Knuth, together with some colleagues and students, studied the behavior of FORTRAN programs, using several different methods of analysis. He concludes ("An Empirical Study of FORTRAN Programs", Software--Practice and Experience): "The program profiles (i.e., collections of frequency counts) which we used in our analyses turned out to be so helpful that we believe profiles should be made available routinely to all programmers by all of the principal software systems." Also, "...our group came to the almost unanimous conclusion that all software systems should provide frequency counts to all programmers, unless specifically told not to do so."

Richard Sites goes so far as to say ("Programming Tools: Statement Counts and Procedure Timings", ACM Sigplan Notices): "Statement counting is the single most useful tool that a programming system can provide to the user."

The Value of a Flow Graph

A flow graph does not contain as much information as a profile, but it can be generated at a fraction of the cost and does not require a running program. It is useful for analyzing program flow, especially during the early stages of coding, before profiling can be attempted. For example, consider any executable FORTRAN statement with a statement number (e.g., 45 CONTINUE). Which statements in the source module may pass control to this statement? There may be any number of such predecessor statements and they may be sprinkled throughout the source module. To find them in an ordinary source-code listing requires an inspection of the entire source module, and even then some may be missed. A flow graph gives this information at a glance.

A flow graph is also useful for preparing test data for a profile. Showing, as it does, all the possible paths of a module, a flow graph

can suggest test data values and ranges for causing particular paths to be traversed.

Running the Profiler

The Profiler takes as input a set of FORTRAN source modules. (What IBM FORTRAN documentation calls "program units" are called "modules" in this description; these may be main programs, block-data programs, subroutines, or functions.) The Profiler parses these modules to extract information relevant to program flow and generates a flow graph. The Profiler then creates a modified version of the source code, known as the target code which is then compiled and executed. The modifications in the target code enable the user program to gather statistics on its own execution. From the data collected and from the flow graph, the Profiler generates a profile.

The dialect of FORTRAN accepted by the Profiler is FORTRAN IV, described in the IBM publication, IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515. This is also the language accepted by the FORTRAN G and H compilers available in MTS.

The Profiler is invoked by the MTS command:

```
$RUN *PROFORT
```

The user controls the Profiler by issuing commands in the Profiler Command Language, described below. One way to issue commands is to build a file of commands and specify that file in the PAR field of the $RUN command.

```
$RUN *PROFORT PAR=filename
```

If the PAR field is omitted, the Profiler reads its commands from *SOURCE* (i.e., defaulting to the user's terminal or the batch input stream). If the PAR field is included, the Profiler executes the commands in the file specified and then reads any further commands from *SOURCE*.

The Profiler is composed of three processors, the Command Processor, the Pre-Execution Processor, and the Execution Processor.

The Command Processor interprets the user's commands and causes the various processors and subprocessors described below to do their jobs in a selective manner in accordance with the user's wishes. It also issues error messages for invalid commands.

Any source module that is to be profiled must at some time pass through the Pre-Execution Processor. The job of this processor is to transform a source module into an object module capable of collecting path frequencies during its own execution.

The Pre-Execution Processor processes each source module in the user's source stream. It completely processes each module according to the user's commands before processing the next module.

The Pre-Execution Processor is composed of five subprocessors:

The Parser reads a source module from the user's source stream and gathers what flow information it can by examining the source code sequentially one statement at a time. It issues error messages for invalid FORTRAN statements.

The Flow Analyzer gathers flow information which requires knowledge of the entire module. It issues warning and error messages regarding program flow in the module.

The Flow Graph Generator writes a flow graph (see Figure 2).

The Code Generator generates a new FORTRAN source module, called a target module, which when compiled and executed will perform exactly as the original source module would if it were compiled and executed, except that the target module will also gather information showing how many times certain critical paths in the module have been traversed. The Code Generator also writes for each target module a set of tables which are required by the Execution Processor.

The Compiler calls the FORTRAN G Compiler to compile the target module produced by the Code Generator.

The job of the Execution Processor is to execute the user's program and to generate profiles from the path frequencies collected.

For each module to be profiled the user must specify the location of an object module produced from a target module and the location of its associated tables. Not every module in a user program need be profiled. For modules that are not to be profiled, the user need only specify the location of object code.

The Execution Processor is composed of three subprocessors:

The Execution Monitor loads and runs the user program. When the user program terminates execution, the user program is unloaded and control passes back to the Execution Monitor.

To minimize execution time, the user program collects frequencies for only certain critical paths of the program. The Frequency Analyzer, using flow information from the tables and the path frequencies collected by the user program, calculates the frequencies for all other program paths, as well as statement frequencies.

The Profile Generator writes a profile (see Figure 3).

The Command Language

At the beginning of a Profiler run, the Profiler is in command mode, which can be recognized by the prefix character "¬". Each command must appear on a separate line. In the following description of the command language the shortest permissible abbreviations are underlined.

Pre-Execution Processor Commands

PREPARE

The PREPARE command alerts the Profiler that the Pre-Execution Processor is to be invoked. It should be followed by commands that specify what the Pre-Execution Processor is to do. These should be followed by the GO command, which causes the Pre-Execution Processor to perform the jobs requested.

From the time that the PREPARE command is issued until the Pre-Execution Processor is finished, the Profiler is in PREP mode, which can be recognized by the prefix characters "PREP¬". The following commands may be issued in PREP mode.

{FLOWGRAPH|NOFLOWGRAPH}={ALL|list}

The FLOWGRAPH command causes a flow graph to be written for each module specified. The term "ALL" means all modules in the source stream. The term "list" means a list of module names separated by commas. The NOFLOWGRAPH command suppresses the writing of flow graphs for each module specified. The default setting is NOFLOWGRAPH=ALL.

Note: The effect of issuing more than one command of this type is cumulative. For example, if a flow graph is wanted of all but modules A and B from a source stream containing 40 modules, F= followed by a list of 38 modules could be specified, or more easily, F=ALL followed by NOF=A,B could be specified.

{CODEGEN|NOCODEGEN}={ALL|list}

The CODEGEN command causes the Code Generator to produce a target module and its associated tables for each module specified. The NOCODEGEN command suppresses the production of target modules and tables for each module specified. The default setting is CODEGEN=ALL.

Note: The effect of issuing more than one command of this type is cumulative.

{COMPILE|NOCOMPILE}={ALL|list}

The COMPILE command causes each module specified to be compiled by the FORTRAN G Compiler. If code has been generated for a module, it is the target module that is compiled. If not, it is the user's

original source module that is compiled. The NOCOMPILE command suppresses compilation of each module specified. The default setting is COMPILE=ALL.

Note: The effect of issuing more than one command of this type is cumulative.

SOURCE=FDname

The SOURCE command assigns a file or device for the user's FORTRAN source code. This is a required command, that is, the Pre-Execution Processor will not run if SOURCE is not assigned. The term "FDname" means "file or device name" and refers to any valid MTS file name, device name, or pseudo-device name. However, the user is warned that the Profiler does not check for invalid assignments such as assigning nonexistent files or assigning inputs to strictly output pseudo-devices (e.g., SOURCE=*PRINT*).

Note: Source code must be in IBM format, however, columns 73 through 80 will be ignored by the Profiler. Source code in other formats (i.e., LINE, LONG, EDIT) may be converted to IBM format with the program *FTN. For a description of FORTRAN source formats, see the section "*FTN Interface" in this volume.

Note: BLOCK DATA modules may appear in the source stream, but as they are not executable, no flow graphs, target code, tables, or profiles will be generated for BLOCK DATA modules, even if requested. The only reason to include BLOCK DATA modules in the source stream is to compile them along with the rest of the source code.

FLOWOUT=FDname

The FLOWOUT command assigns a file or device for the flow graph output. If flow graphs have been requested and FLOWOUT is not specified, the default file is -FLOWGRAF. If -FLOWGRAF does not exist, it will be created. If it already exists, it will be emptied before output is written.

BOXES={ON|OFF}

When BOXES=ON, flow graphs are written with each executable statement and its flow information set off by boxes in order to make reading easier (see Figure 2). When BOXES=OFF, flow graphs are written without the boxes. The default setting is BOXES=ON.

INDICATORS={ON|OFF}

The ISNs (Internal Statement Numbers) in the PRED and SUCC columns of flow graphs (see Figure 2) are sometimes preceded by special symbols called indicators. An asterisk indicates that the path in question is the result of normal sequencing from one statement to the next. A pound sign indicates that the path in question is a

DO-loop return path. When INDICATORS=ON, these indicators appear in flow graphs. When INDICATORS=OFF, they are omitted. The default setting is INDICATORS=ON.

TARGET=FDname

The TARGET command requests that any target modules generated be written on the file or device specified.

Generally, once a target module is compiled it is not interesting to look at or to save. Therefore, whenever a source module is selected for both code generation and compilation, the target module is generated in virtual memory and passed directly to the Compiler. Unless TARGET is specified, it is not saved.

However, if code generation is requested and compilation is not (e.g., CODEGEN=ALPHA, NOCOMPILE=ALPHA), the target module is saved on the assumption that it will be wanted for later compilation. In this case, the default file is -TARGET. If -TARGET does not exist, it will be created. If it already exists, it will be emptied before output is written.

TABLES=FDname

The TABLES command assigns a file or device for the tables built by the Code Generator. If TABLES is not assigned, the default file is -TABLES. If -TABLES does not exist, it will be created. If it already exists, it will be emptied before output is written.

OBJECT=FDname

The OBJECT command assigns a file or device for the object code generated by the Compiler. If OBJECT is not assigned, the default file is -OBJECT. If -OBJECT does not exist, it will be created. If it already exists, it will be emptied before output is written.

GO

If the Profiler is in PREP mode, the GO command invokes the Pre-Execution Processor. When the Pre-Execution Processor is finished, the Profiler reenters command mode and all I/O assignments and options are reset to their default values, if any.

Execution Processor Commands

EXECUTE

The EXECUTE command alerts the Profiler that the Execution Processor is to be invoked. It should be followed by commands that specify what the Execution Processor is to do. These should be followed by the GO command, which causes the Execution Processor to perform the jobs requested.

From the time that the EXECUTE command is issued until the Execution Processor is finished, the Profiler is in EXEC mode, which can be recognized by the prefix characters "EXEC¬". The following commands may be issued in EXEC mode.

OBJECT=FDname

The OBJECT command specifies the location of the object program to be run by the Execution Processor. The default is the most recent previous setting, whether set in PREP or EXEC mode, explicitly or by default.

TABLES=FDname

The TABLES command specifies the location of the tables associated with the object program to be run. The default is the most recent previous setting, whether set in PREP or EXEC mode, explicitly or by default.

PROFILE=FDname

The PROFILE command assigns a file or device for the profiles written by the Profiler. If PROFILE is not assigned, the default file is -PROFILE. If -PROFILE does not exist, it will be created. If it already exists, it will be emptied before output is written.

BOXES={ON|OFF}

When BOXES=ON, profiles are written with each executable statement and its flow and frequency information set off by boxes in order to make reading easier (see Figure 3). When BOXES=OFF, profiles are written without the boxes. The default setting is BOXES=ON.

INDICATORS={ON|OFF}

The ISNs (Internal Statement Numbers) in the PRED and SUCC columns of profiles (see Figure 3) are sometimes preceded by special symbols called indicators. An asterisk indicates that the path in question is the result of normal sequencing from one statement to the next. A pound sign indicates that the path in question is a DO-loop return path. When INDICATORS=ON, these indicators appear in profiles. When INDICATORS=OFF, they are omitted. The default setting is INDICATORS=ON.

lio=FDname

"lio" refers to any of the MTS logical I/O units (i.e., SCARDS, SPRINT, SPUNCH, GUSER, SERCOM, or 0 through 19). This command allows the user to specify I/O assignments for the execution of the user program, in the same manner as on a $RUN command.

The usual MTS defaults apply, that is:

```
SCARDS=*SOURCE*
SPRINT=*SINK*
SPUNCH=*PUNCH*, if in batch mode and if the CARDS global
       parameter has been specified on the $SIGNON command.
GUSER=*MSOURCE*
SERCOM=*MSINK*
```

Also, the usual FORTRAN I/O Library defaults apply, that is:

```
5=*SOURCE*
6=*SINK*
```

PAR=parstring

The PAR command allows the user to specify a PAR field for the user program, in the same manner as on a $RUN command. There is no default setting.

The recommended method of accessing this PAR field from the user program is to call the PAR subroutine. Calling GUINFO for the PAR field does not work; it will access the PAR field from the $RUN *PROFORT command. For descriptions of PAR and GUINFO, see MTS Volume 3, System Subroutine Descriptions.

Another way to access the user PAR field requires that the main routine of the user program be a subroutine with a single argument designed to hold a character string. The PAR field will be found in that string with the length of the PAR field as an integer in the first two bytes and the PAR string itself beginning in the third byte. In the following example, if a PAR field is present, it is written on SERCOM.

```
SUBROUTINE MAIN (PARFLD)
LOGICAL*1 PARFLD(257),PBYTE(2)
INTEGER*2 PARLEN/0/
EQUIVALENCE (PARLEN,PBYTE(1))
PBYTE(2)=PARFLD(2)
IF (PARLEN.NE.0) CALL SERCOM (PARFLD(3),PARLEN,0)
```

GO

If the Profiler is in EXEC mode, the GO command invokes the Execution Processor. When the Execution Processor is finished, the Profiler reenters command mode and all I/O assignments and options are reset to their default values, if any.

Commands That Can Be Issued in Any Mode

COMMANDS=FDname

The COMMANDS command assigns an alternate input path for commands to the Profiler. If COMMANDS is assigned, the Profiler reads its commands from the file/device specified until an end of file is

sensed, or until a command error is detected, at which time the setting of COMMANDS reverts to *SOURCE*. If COMMANDS is not assigned, the Profiler reads its commands from *SOURCE*.

MESSAGES=FDname

The MESSAGES command assigns an additional output path for the messages printed by the Profiler. Messages are always printed on *SINK*. If MESSAGES is assigned, then Profiler messages are also written on the file/device specified.

$mts-command

Any command beginning with a dollar sign will be passed on to MTS for immediate execution, followed by return to the command stream.

MTS

The MTS command returns control to MTS command mode. The user may issue MTS commands and return to the Profiler by issuing the MTS command $RESTART.

CLEAR

If the Profiler is in PREP or EXEC mode, the CLEAR command returns the Profiler to command mode and resets all I/O assignments and options to their default values, if any. This command may be used to restart the Profiler from the beginning. If the Profiler is already in command mode, the command is ignored.

STOP

The STOP command unloads the Profiler and returns control to MTS command mode. It should be used to terminate the Profiler.

Error Messages

The Profiler reports four classes of errors.

Command errors

If the user enters an erroneous command, it is reported and the setting of COMMANDS reverts to *SOURCE*. The user may then enter a new command.

Source errors

These are errors detected by the Parser in the user's FORTRAN source code. The Parser does not check for all possible source errors. It leaves this job to the Compiler. Since the Profiler is only concerned with the flow information inherent in a source module, it parses only enough of each statement to extract such

information. For example, it parses assignment statements only to the point where it can identify them as assignment statements. Any errors to the right of the equals sign remain undetected until compilation.

The Profiler should not be used to flush out syntax errors. In fact, it is best to use the Profiler with programs that not only compile, but run to completion.

When the Parser finds a source error, it stops parsing the faulty statement, but continues parsing the rest of the source module in order to report any other errors it might encounter. After that, nothing further is done with that module, no matter what the user has requested. There is no point going any further with a module that will not make it past the Compiler. The Profiler will, however, continue to process the remaining modules in the source stream.

Flow errors

These are errors detected by the Flow Analyzer. Since the Profiler must construct the complete flow graph of a source module, it has very good knowledge of flow errors, and the messages in this category tend to be better than those reported by the Compiler.

When a flow error is found, the Flow Analyzer continues its analysis of the module in order to report all flow errors it can find. After that, the only user request concerning that module that will be honored is a request for a flow graph. The Profiler will, however, continue to process the remaining modules in the source stream.

A number of flow warnings are also issued for situations that are legal, but suspicious. For example, a warning is issued if a statement number is defined, but there are no references to it. Flow warnings do not inhibit further processing of the module.

Compiler errors

When the Profiler reports that there are Compiler errors, the user can $COPY -COMPERR to see what they are.

If any source, flow, or compiler errors are reported for any of the modules in the source stream and this is followed by an EXECUTE command, then the user is warned and prompted for an 'OK' to continue with execution.

References

B.W. Kernighan and P.J. Plauger, in Software Tools, 316-318, Addison-Wesley, 1976.
D.E. Knuth, "An Empirical Study of FORTRAN Programs," Software -- Practice and Experience, 1:105-133, 1971.
G. Lyon and R.B. Stillman, "Simple Transforms for Instrumenting FORTRAN Decks," Software -- Practice and Experience, 5:347-358, 1975.
G.J. Myers, The Art of Software Testing, John Wiley & Sons, 1979.
R.L. Sites, "Programming Tools: Statement Counts and Procedure Timings," ACM Sigplan Notices, 13:12:98-101, Dec., 1978.
G. Waldbaum, "Tuning Computer Users' Programs," IBM Computer Science Research Report, Dec., 1978.

Example Runs

In the example below, the Pre-Execution Processor processes all the source modules in the file PROGRAM.S, writing the tables for each module on the file -TABLES and object code on the file -OBJECT. No flow graphs are written. The target modules are passed to the Compiler in virtual memory and are not written.

```
PREPARE
SOURCE=PROGRAM.S
GO
```

The example below is the same as the previous example, except that tables are written on file A, object code is written on file B, and a flow graph for each module is written on file -FLOWGRAF.

```
PREP
S=PROGRAM.S
T=A
O=B
F=ALL
GO
```

The example below is also like the first example, except that the Execution Processor is also invoked. The object code in -OBJECT is run, using the tables in -TABLES, and a profile for each module is written on file PROFILE without indicators or boxes.

```
PREP
S=PROGRAM.S
GO
EXECUTE
P=PROFILE
INDIC = OFF
BOXES = OFF
GO
```

Suppose a user wants to run a large program, taking profiles on the printer of just two of its subroutines. Object code for the entire program is in file PROG.O. Source code for the two modules to be profiled is in file PROG.S(101,149) and PROG.S(279,315). The program requires that I/O unit 1 be assigned for input.

```
PREP
SOURCE=PROG.S(101,149)+(279,315)
OBJECT=OBJ
TABLES=TABLES
GO
EXEC
OBJECT=OBJ+PROG.O
PROFILE=*PRINT*
1=INFILE
GO
```

Note the command OBJECT=OBJ+PROG.O. Both OBJ and PROG.O contain object modules for the two subroutines to be profiled. The ones in OBJ, derived from the target modules, are the ones the user wants loaded. So, OBJ appears first in this command, because if the loader encounters more than one module with the same name, it loads only the first such module. However, the user should be certain, when setting up commands like this, that the entry point is correctly determined. (For the rules on entry point determination, see "The Dynamic Loader, Appendix A" in MTS Volume 5, System Services.)

Suppose the user wants to generate more profiles for the program of the previous example using two additional sets of input data. The following command sequence may be used.

```
EXEC
OBJ=OBJ+PROG.O
TAB=TABLES
P=*PRINT*
1=INFILE2
GO
EXEC
P=*PRINT*
1=INFILE3
GO
```

Note that it is not necessary to assign OBJECT or TABLES for the second execution, as they default to the previous usage.

A user has a source program in the file PROG.S and wants to compile and run the whole program, but wants to profile only subroutine SORT as it works in the context of the whole program. The following command sequence may be used.

```
PREP
S=PROG.S
NOCODEGEN=ALL
CODEGEN=SORT
O=PROG.O
T=PROG.TAB
GO
EXEC
P=*PRINT*
GO
```

Note that for subroutine SORT, the target module is compiled, while for the rest of the source stream, the original source modules are compiled.

October 1983

MISCELLANEOUS FORTRAN PROGRAMS

The following public file descriptions are taken from MTS Volume 2, Public File Descriptions. These are programs which may be of use to FORTRAN users.

MTS 6: FORTRAN in MTS

October 1983

October 1983

*DAVE

Contents: The data-flow analyzer for FORTRAN programs.

Use: The analyzer is invoked by the $RUN command.

Program Key: *EXEC

Logical I/O Units Referenced:

SCARDS - the FORTRAN program to be analyzed.
SPRINT - output from the analyzer.
SERCOM - error messages and program comments.
GUSER - responses to prompting messages.

Description: *DAVE is a software tool for gathering information about global data flow in FORTRAN programs, and for identifying the anomalous use of data in these programs. *DAVE is a static analysis tool, meaning that *DAVE gathers information about the subject program without executing it. *DAVE does not require modification of the subject program, nor does it require intervention by the user during execution. Only an initial setting of parameters that control the output is required.

For the complete details of using the *DAVE analyzer program and the *DAVE.GENLIB subroutine library, see Computing Center Memos 394 and 402.

MTS 6: FORTRAN in MTS

October 1983

October 1983

*FTNTIDY

Contents: The FORTRAN "tidying" program.

Purpose: To tidy FORTRAN source programs into an easily readable format and/or to produce cross-reference listings.

Use: The program is invoked by the $RUN command.

Program Key: *FTNTIDY

Logical I/O Units Referenced:

SCARDS - one or more FORTRAN source programs to be tidied and/or cross-referenced.
SPRINT - the listing of the program(s) and the cross-references.
SPUNCH - the tidied FORTRAN source program(s).
SERCOM - severe error comments.

Parameters: The following parameters may be specified in the PAR field of the $RUN command. The parameters must be separated by commas or blanks. In case of conflicting parameters, the rightmost parameter takes precedence. Some of the parameters, as indicated below, may be negated by prefixing them with "NO", "N", "-", or "¬". Alternatively, these same parameters may be written as parm=ON or parm=OFF, where "parm" is the parameter name. Thus, SOURCE is the same as SOURCE=ON, and NOSOURCE the same as SOURCE=OFF. The minimum acceptable abbreviation for each parameter is underlined. No embedded blanks are allowed within a parameter.

[NO]SOURCE — SOURCE produces the listing of the original FORTRAN source program on SPRINT. The default is SOURCE if no tidying is to be done; otherwise, it is NOSOURCE.

ISN/MTSLNR — Normally, the cross-reference listings produced by FTNTIDY refer to internal statement numbers (ISNs). If MTSLNR is specified, the listings will use MTS line numbers. If tidying is being done, the MTS line numbers will refer to the file or device assigned to SPUNCH; otherwise the line numbers will refer to the file or device assigned to SCARDS. Note that if implicit or explicit concatenation is used, MTS line numbers may not be unique.

[NO]XREF — XREF causes all cross-reference dictionaries to be printed on SPRINT. NOXREF suppresses the cross-references. The default is normally XREF. If tidying is to be done and SPRINT defaults to a terminal, NOXREF is assumed.

[NO]LBLXREF — LBLXREF suppresses printing of all cross-reference dictionaries except the statement label dictionary. The default is NOLBLXREF.

FORMAT={IBM|EDITED|LINE|LONG} — FORMAT specifies which of the three source statement formats should be expected. The available formats are IBM, EDITED, LINE, and LONG. These may be abbreviated to I, E, L, and LO, respectively. The default is EDITED. For the description of these formats, see Source Statement Formats in the section "*FTN Interface" in this volume.

[NO]BCD — BCD specifies that the source is in Binary Coded Decimal (026 Keypunch). The default is NOBCD (i.e., EBCDIC).

LINECNT=n — LINECNT specifies the number of lines per page to be printed. The range is 2 to 32767; the default is 60.

ERRMAX=n — ERRMAX specifies the maximum number of errors FTNTIDY may tolerate while it is processing a subprogram. If there are more than "n" errors, FTNTIDY will terminate the processing. The default is 25.

[NO]DECK — DECK causes tidied FORTRAN source statements to be produced and written on SPUNCH. The default is DECK if SPUNCH is assigned on the $RUN command; otherwise it is NODECK.

The parameters that follow control FTN TIDY's tidying if DECK has been specified explicitly or by default. If tidying is not being done, these parameters are ignored.

[NO]LIST — LIST causes a listing of the tidied source program to be produced on SPRINT. The default is normally LIST. If SPRINT defaults to a terminal, NOLIST is assumed.

[NO]RELABEL — RELABEL causes all statement numbers to be renumbered so that they increase in ascending order. NORELABEL retains the original statement numbers. The default is RELABEL.

START=n — The number "n" will be used as the first statement number in the relabeling process. The START default is 10. START has no effect if NORELABEL has been specified.

INCR=n — The number "n" is used as increment between two successive statement numbers if RELABEL is in effect. The default is INCR=10. INCR has no effect if NORELABEL has been specified.

[NO]FMTMOVE — FMTMOVE causes FTNTIDY to collect all FORMAT statements and place them at the end of the program. The default is NOFMTMOVE.

[NO]SPACE — SPACE causes FTNTIDY to remove all irrelevant blanks and insert single blanks in the tidied FORTRAN source to improve read ability. The default is SPACE. This parameter may also be specified as SPACE={ON|OFF}.

The NOSPACE parameter overrides the spacing option for single statement or a list of statements; it is specified as

```
NOSPACE=stmt
NOSPACE=(stmt1,stmt2,...)
```

where "stmt" is a FORTRAN statement type (e.g., FORMAT, DOUBLEPRECISION). For example, NOSPACE=FORMAT will suppress the spacing option for FORMAT statements.

[NO]INDENT — INDENT causes statements within DO loops to be indented according to the nesting level. Continuation of statements beginning with FORTRAN keywords are also indented. INDENT may also be expressed as INDENT=n, where "n" specifies the number of additional columns statements are to be indented for each level of nesting depth. "n" must lie in the range (0,10), inclusively. The default is INDENT=2.

[NO]DOCOMMENT DOCOMMENT causes FTNTIDY to insert blank comments before and after DO loops. The default is NODOCOMMENT.

RTMARG=n A right margin of "n" may be specified to limit the length of tidied statements. "n" should lie in the range 50 to 72, inclusive; the default is 72. FTNTIDY always makes exceptions for Hollerith and quoted strings, for which the right margin is 72.

[NO]SEQ SEQ causes FTNTIDY to place sequence numbers in columns 73-80 of the tidied source program. The default is NOSEQ.

LBLJUST={LEFT|RIGHT}
Labels in columns 1-5 of the tidied source program will be either left- or right-justified. The default is RIGHT.

[NO]HOLQUOTE HOLQUOTE converts Hollerith fields (strings preceded by "nH", where "n" is the number of characters in the string) into strings enclosed in apostrophes. NOHOLQUOTE leaves literal constants in their original form. The default is HOLQUOTE. This parameter may also be specified as HOLQUOTE={ON|OFF}.

CONTCHAR='c' The single character "c" (enclosed within primes) specifies the continuation character that is to be inserted in column 6 of continuation lines. If "c" is blank or zero, the sequence 1, 2, 3, ..., 9, *, 1, 2, ... will be inserted in column 6. For example, CONTCHAR= '*' specifies that column 6 of continuation lines is set to "*". The default is blank.

Description: FTNTIDY may be used to tidy FORTRAN source programs and/or to produce a cross-reference of the variables, statement numbers, functions and subroutines, and the FORTRAN logical I/O units used. By default, if the MTS logical I/O unit SPUNCH is not assigned on the $RUN command, FTNTIDY produces a listing of the source followed by the cross-reference listings.

If SPUNCH is assigned, then FTNTIDY will tidy the FORTRAN source. If SPRINT does not default to a terminal, the tidied source is listed on SPRINT followed by the cross-reference listings. The following are the default tidying operations performed by FTNTIDY:

(1) All statement numbers are renumbered in ascending order: 10, 20, 30, etc.
(2) All blanks except those appearing in Hollerith fields and quoted strings are removed. Single blanks are then inserted to improve readability.
(3) Statements within DO loops are indented according to their nesting level.
(4) Continuation statements beginning with FORTRAN keywords are indented.

Optionally, the user may override these defaults by specifying parameters, such as NORELABEL, to retain the original statement numbers. The user may also specify other tidying features, such as moving all FORMAT statements to the end of a program (FMTMOVE), or inserting blank comments before and after DO loops (DOCOMMENT).

If XREF is in effect, FTNTIDY produces four cross-reference dictionaries on SPRINT. These are:

(1) subprograms, which consist of all subroutines, functions, and entries
(2) variables
(3) statement numbers
(4) logical I/O units (excluding variables)

For the first two dictionaries, names are arranged in alphabetical order; for the last two, labels and units are printed in ascending sequence.

A brief explanation for TYPE, ATTR (attributes), and REFERENCES appears at end of each cross-reference listing. TYPEs as shown in the first three dictionaries are:

L*1	LOGICAL of length 1
L*4	LOGICAL of length 4
I*2	INTEGER of length 2
I*4	INTEGER of length 4
R*4	REAL of length 4
R*8	REAL of length 8
R*16	REAL of length 16
C*8	COMPLEX of length 8
C*16	COMPLEX of length 16
C*32	COMPLEX of length 32
CHARS	CHARACTER
GEN.	GENERIC function
N.L.	NAMELIST variable
FMT	FORMAT statement number

TYPEs enclosed within parentheses indicate that these variables were implicitly declared.

Attributes in the subprogram and variable dictionaries are:

SUBR	Subroutine
FCN	Function
ENTRY	Entry
S.F.	Statement function
EXT.	External
ARRAY	Variable array

There is a special subprogram name <EXIT>, used to refer to either RETURN or STOP statements.

If variables appear in COMMON statements, the associated common block names are always shown. "//" is used for a blank common block.

If RELABEL was in effect, the statement label dictionary also shows the original statement numbers under the heading "ORIG". ISN (or MTS line number) defining the statement numbers are printed under the heading "DEFN". If a statement number is undefined, "*****" is printed instead.

For all four dictionaries, references to MTS line numbers are printed with a decimal point, while references to internal statement numbers (ISNs) are printed without a decimal point. In addition, FTNTIDY may insert a special character to the right of each reference as follows:

*	A variable or a function is changed either through an assignment, READ, ASSIGN statement, or use as a DO index.
?	A variable may be changed because it is used as a simple argument to a subroutine or function.
D	A subprogram is defined by the SUBROUTINE, FUNCTION, ENTRY, or EXTERNAL statement. A statement function is defined by the statement function definition. A variable is declared in a type or DIMENSION statement. For units, "D" stands for DEFINE FILE statements.
E	A variable appears in an EQUIVALENCE statement.
C	A variable appears in a COMMON statement.
R	A unit appears in a READ statement.
W	A unit appears in a WRITE, PRINT, or PUNCH statement.

M A unit appears in other I/O statements defining a motion (e.g., REWIND, BACKSPACE, ENDFILE, FIND, or WAIT).

Examples: In the following example, a cross-reference listing is generated for the FORTRAN program in the file TEST.

```
$RUN *FTNTIDY SCARDS=TEST

  MTS      INTERNAL       ****  F T N T I D Y ****
 LINE NO.  STMT NO.       INPUT LISTING

  100.          1          FUNCTION DRSINH(DX)
  101.          2          IMPLICIT REAL*8 (D)
  102.          3          COMMON DLOG2,MSG(4)
  103.          4    1000  DCON = 1.0D0
  104.          5          GO TO 132
  105.               C
  106.          6          ENTRY DRCOSH(DX)
  107.          7    993   DCON = -1.0D0
  108.          8          IF(DX.GE.1.0D0) GOTO 132
  109.               C     FAILURE  -   RETURN
  110.          9          DRSINH = 0.0D0
  111.         10          WRITE (6,887) MSG
  112.         11    887   FORMAT ( '0', 4A4)
  113.         12          RETURN
  114.               C
  115.         13    132   DY = DABS(DX)
  116.         14          IF (DY.GT.1.0D8) GO TO 1070
  117.         15          IF (DY.LT.1.0D-4) GOTO 2196
  118.               C     NORMAL CASE
  119.         16          DW = DLOG(DY+DSQRT(DY**2+DCON))
  120.         17          GO TO 111
  120.2             C     SMALL DY
  120.4        18    2196  DW = DY - DY**3/6.0D0
  120.6        19          GOTO 111
  121.              C     LARGE DY
  122.         20    1070  DW = DLOG(DY) + DLOG2
  123.         21    111   DRSINH = DSIGN(DW,DX)
  124.         22          RETURN
  125.         23          END

              ***   SUBPROGRAM  DICTIONARY  ***

    NAME    TYPE  ATTR      REFERENCES

    DABS    R*8   FCN       13
    DLOG    R*8   FCN       16     20
    DRCOSH (R*8)  ENTRY      6D
    DRSINH (R*8)  FCN        1D     9*    21*
    DSIGN   R*8   FCN       21
    DSQRT   R*8   FCN       16
    <EXIT>        SUBR      12     22
```

```
                ***   VARIABLE  DICTIONARY  ***

NAME   TYPE  ATTR  COMMON  REFERENCES

DCON   (R*8)                    4*     7*    16
DLOG2  (R*8)        //          3C    20
DW     (R*8)                   16*    18*    20*    21?
DX     (R*8)                    1      6      8     13?
                               21?
DY     (R*8)                   13*    14     15     16
                               18     20?
MSG    (I*4) ARRAY //           3C    10

          ***  STATEMENT LABEL DICTIONARY  ***

 LABEL  DEFN     TYPE           REFERENCES

   111    21                    17     19
   132    13                     5      8
   887    11     FMT            10
   993     7
  1000     4
  1070    20                    14
  2196    18                    15

          ***  LOGICAL I/O UNIT DICTIONARY  ***

  UNIT               REFERENCES

     6    10W
```

October 1983

*FTNTOPL1

Contents: The FORTRAN-IV-to-PL/I language conversion program (version 1, modification level 0).

Use: The program is invoked by the $RUN command.

Program Key: *FTNTOPL1

Logical I/O Units Referenced:

SCARDS - Source for the FORTRAN program to be converted.
SPRINT - FORTRAN and PL/I source listings and diagnostic messages.
SPUNCH - 80-character deck output of the resultant PL/I program.

Description: The program attempts to convert source programs written in FORTRAN IV to their PL/I equivalents. It detects and flags FORTRAN IV statements that have no PL/I equivalent or that cannot be meaningfully or unambiguously translated into PL/I statements. Conflicts between the use of FORTRAN and PL/I library subroutines are also noted.

It should be noted that all references to logical unit 6 are replaced in the PL/I program by #06, which is automatically declared with the PRINT attribute. In general, references to logical unit "n" are replaced by PL/I file names of the form, #0n.

A complete description of this conversion program is given in the IBM publication, FORTRAN IV to PL/I Language Conversion Program, form GC33-2002.

Parameters: If parameters are specified, they must appear in the first line read from SCARDS. Column one of this line must contain a percent sign "%"; there may be no blanks between the percent sign and the last character in the parameter field; parameters must be separated by commas. The parameters to the program are presented below. Abbreviations are underlined.

BCD / EBCDIC Default: EBCDIC

Specifies the character code of the FORTRAN source program and, consequently, that of the resultant PL/I program.

BLKZR or BZ / NOBLKZR or NBZ Default: NOBLKZR

Specifies whether the external form of numeric input data must be processed during execution of the PL/I program by the library subroutine, LBLNK.

CHAR48 or C48 / CHAR60 or C60 Default: CHAR60

Specifies the PL/I character set to be used in the converted program.

DECK / NODECK or ND Default: NODECK

Specifies whether the PL/I program is to be punched on SPUNCH.

EXTREF / NOEXTREF or NE Default: NOEXTREF

Specifies whether name changes in the FORTRAN source program are to be listed on SPRINT.

SOURCE / NOSOURCE or NS Default: SOURCE

Specifies whether the FORTRAN source program is to be listed on SPRINT.

Example:

```
$RUN *FTNTOPL1 SPUNCH=PLS
%DECK
[Program 1]
[Program 2]
   .
   .
   .
$ENDFILE
```

In the above example, the FORTRAN programs in the input stream are converted and written to the file PLS. The %DECK parameter specifies that the resulting PL/I programs are punched on SPUNCH.

October 1983

*PFORT

Contents: The PFORT Verifier.

Purpose: To check FORTRAN programs for adherence to PFORT, a portable subset of the 1966 version of American National Standard FORTRAN.

Use: The program is invoked by the $RUN command.

Program Key: *PFORT

Logical I/O Units Referenced:

SCARDS - Input to the Verifier. This should take the form of one or more FORTRAN source program units to be verified. If global analysis is requested, the sequence of source program units should comprise a single executable FORTRAN program. An input line containing a period in column 1 will cause the Verifier to stop processing input. However, the preferred method of specifying the end of the input is to use one of the usual end-of-file mechanisms supported by MTS.

SPRINT - Output from the Verifier. This may be program listings, messages indicating PFORT violations, symbols tables, cross references, or global analysis.

Parameters: The following parameters or their negations may be specified in the PAR field of the $RUN command. Parameters must be separated by blanks or commas. The minimum acceptable abbreviation for each parameter is underlined. The parameters are treated from left to right, so in case of conflicting parameters, the rightmost one will be put into effect. Any parameter may be negated by prefixing it with NO, N, -, or ¬. All of the parameters below are active by default.

LIST Print a source listing for each program unit.

SYMBOLS Print a symbol table for each program unit.

XREF Print cross references with each symbol table. This option will be honored only if SYMBOLS is active.

GLOBAL Perform and print a global analysis of the input source program. The Verifier itself may deactivate this option if it encounters serious

errors during the processing of the input. Once deactivated, whether by the user or by the Verifier, the GLOBAL option cannot be reactivated.

Parameters may also be specified by including a source line with a C in column 1 and an asterisk in column 2, followed by parameters anywhere in columns 3 to 72. This permits the user to turn the output options on or off locally.

Example:
```
C*  LIST
       SUBROUTINE A
        ...
       END
C*  NOLIST
       SUBROUTINE B
        ...
       END
C*  LIST
       SUBROUTINE C
        ...
```

In this example, selected program units are LISTed. Such local parameters embedded in the source may be safely passed on to any of the FORTRAN compilers, which will treat them as ordinary FORTRAN comments.

Description: The PFORT Verifier produces a statement by statement listing of each program unit, followed by a symbol table which lists attributes and cross references for all the symbols in that program unit.

If a violation of the PFORT standard is discovered, an error message, preceded by three asterisks, is printed directly below the statement at which the error occurred. If the NOLIST option is in effect, the error message is accompanied by the program unit name and internal statement number of the statement in error.

The global structure of the program is presented in an alphabetized list of all its program units, showing for each program unit its arguments, its common blocks, the program units it calls, and the program units which call it. A table of global common block definitions is produced.

For the complete description of the PFORT Verifier, see Computing Center Memo 406, "*PFORT."

October 1983

*RATFOR

Contents: A FORTRAN preprocessor program.

Use: To allow control structures for FORTRAN programs.

Program Key: *RATFOR

Logical I/O Units Referenced:
SCARDS - structured FORTRAN input.
SPRINT - structured program listing.
SPUNCH - processed standard FORTRAN source output.

Description: RATFOR (Rational FORTRAN) is a FORTRAN language preprocessor developed by Kernighan and Plauger and described in the publication

> Software Tools, by Brian W. Kernighan and P. J. Plauger, Addison-Wesley, 1976

RATFOR was developed for the purpose of overlaying more modern and commonly accepted control structures and features on FORTRAN, making it a more palatable and versatile tool to utilize in the solution of programming tasks that for one reason or another require the use of FORTRAN. The language description as well as the preprocessor itself are described in the book and the preprocessor as implemented on MTS is very straight forward to use.

The structured FORTRAN program is read by the translator from logical I/O unit SCARDS and the standard FORTRAN source code is produced on logical I/O unit SPUNCH. A listing is produced on logical I/O unit SPRINT in batch mode and also in terminal mode if SPRINT is explicitly assigned to a file or device. The resulting standard FORTRAN output is ready for processing by a standard FORTRAN compiler.

Example: $RUN *RATFOR SCARDS=RATPROG SPUNCH=FORTPROG

> In the above example, the structured FORTRAN program in the file RATPROG is processed into a standard FORTRAN program and written into the file FORTPROG.

October 1983

October 1983

EXCEPTIONAL CONDITIONS

Under some conditions when using *FTN, *FORTRANG, *FTNGTEST, *FORTRANH or *FORTRANVS, one of the following 15 program-interrupt error messages may be produced before the FORTRAN I/O monitor gains control. These messages include the name of the module where the error was first detected. Some of the errors (notably significance, integer overflow, floating-point underflow) are normally "masked-off", that is, if the error condition occurs, it is ignored and execution continues.

Since the error messages do not always provide enough information to correct the error, it may be advisable to use SDS or switch to *IF or *WATFIV to track down the cause of the problem.

The error messages are as follows:

Integer overflow in routine nnnn at hexadecimal displacement +xxx.

> If this exception is masked on, an error will error for i+j, i-j, -j, IABS(j), and arithmetic shift operations that produce a result in the range -2^{31} to $2^{31}-1$, Generally, i*j will not produce an error; however sometimes i*j is reduced to an arithmetic shift, which will cause an error. Normally, this exception is masked off.

Integer division by zero in routine nnnn at hexadecimal displacement +xxx.

> An operation of i/j or MOD(i,j) was performed where "j" was zero, or an overflow occurred during a conversion from decimal data to binary data (CVB instruction). Normally, this exception is masked on.

Floating-point overflow in routine nnnn at hexadecimal displacement +xxx.

> A floating point operation resulted in a number that was too big. Normally, this exception is masked on.

Floating-point divide by zero in routine nnnn at hexadecimal displacement +xxx.

> A division involving REAL numbers was performed where the divisor is zero.

Significance exception in routine nnnn at hexadecimal displacement +xxx.

In an operation of addition or subtraction involving REAL numbers, under some conditions significant digits will be lost. If the mask bit is set on (which it normally is not), then this error condition will occur.

Floating-point underflow exception in routine nnnn at hexadecimal displacement +xxx.

In processing a REAL operation, the exponent was formed which was too small. Normally zero is returned as a result, but if the mask for this interrupt is set on, the error message is produced.

Addressing exception in routine nnnn at hexadecimal displacement +xxx.

Protection exception in routine nnnn at hexadecimal displacement +xxx.

These are caused by improper use of an address, most commonly a bad subscript in an array reference. It also can be caused by improper parameter passing between subprograms.

Operation exception - probably caused by exceeding the dimensions of an array in routine nnnn at hexadecimal displacment +xxx.

Privileged operation exception - probably caused by exceeding the dimensions of an array in routine nnnn at hexadecimal displacement +xxx.

Execute exception - probably caused by exceeding the dimensions of an array in routine nnnn at hexadecimal displacement +xxx.

Data exception in routine nnnn at hexadecimal displacement +xxx.

Decimal overflow in routine nnnn at hexadecimal displacement +xxx.

Decimal divide by zero in routine nnnn at hexadecimal displacement +xxx.

These errors normally will not occur in programs written entirely in FORTRAN. However, they may occur in two cases:

(a) part of the program was overwritten with illegal data, or
(b) an erroneous branch occurred within an assigned GOTO statement.

Either condition usually will result in an immediate error, often an operation, addressing, protection, or specification exception.

Specification exception - probably caused by subroutine with wrong type or by bad alignment in common in routine nnnn at hexadecimal displacement +xxx.

This error can be caused by problems with COMMON or parameter passing between subprograms, by an erroneous branch within an assigned GOTO statement, or by overwriting instructions with data.

October 1983

October 1983

INTRODUCTION TO DEBUG MODE FOR FORTRAN

The Symbolic Debugging System (SDS) is a conversational facility for testing and debugging programs. This facility was originally provided for assembly language programs, but it has now been extended to include FORTRAN programs. Using SDS, the user may initiate the execution of a program and monitor its performance by displaying or modifying variables at strategic points in the program. This section provides a brief introduction to the debug mode command language for FORTRAN users. A small sample FORTRAN program is given to illustrate the use of SDS. The complete description of SDS is given in MTS Volume 13, The Symbolic Debugging System.

Figure 1 is a sample program to compute the mean and standard deviation of an array of real numbers. The program consists of three sections: the main program MAIN which reads in the data values and prints the final results, the subroutine CALC which computes the desired quantities, and a blank-named COMMON section which contains the data array. In FORTRAN, the main program always has the name MAIN unless it is explicitly specified otherwise during the compilation.

This program is compiled by the FORTRAN-G compiler in *FTN using the MTS command

```
$RUN *FTN SCARDS=MEANPROG SPUNCH=MEAN PAR=TEST
```

The source for the program is read from the file MEANPROG and the compiled object module is written into the file MEAN. The TEST parameter must be specified when use of SDS is expected in order to have the FORTRAN compiler produce SYM (symbol table) records in the object module. These symbol table records are used by SDS and are necessary to enable the user to debug his program symbolically.

The most common method of invoking SDS for debugging this sample program is with the MTS command

```
$DEBUG MEAN
```

The DEBUG command is the same as the MTS RUN command in the manner in which logical I/O units and the parameter field are specified. Here it is assumed that the program uses logical I/O unit 5 for reading the input data and logical I/O unit 6 for printing the output results. For the present purpose of debugging this program interactively, all input test data will be entered from the terminal (*SOURCE*) and all output results will printed on the terminal (*SINK*). If the user wishes to assign these units to files, he may specify them on the DEBUG command, e.g.,

```
$DEBUG MEAN 5=INPUTFILE 6=OUTPUTFILE
```

SDS signals its readiness to accept a command by printing the prefix character "+" in column one. This prefix character precedes all SDS messages and diagnostics.

When the program has been successfully loaded, the message

```
+READY
+
```

is printed, at which point SDS is ready to accept its first debug command.

```
0001              DIMENSION DATA(50)
0002              COMMON DATA,N
0003              REAL MEAN
0004            1 WRITE(6,100)
0005          100 FORMAT(' ENTER NUMBER OF DATA POINTS')
0006              READ(5,101) N
0007          101 FORMAT(I3)
0008              WRITE(6,102)
0009          102 FORMAT(' ENTER DATA POINTS')
0010              READ(5,103) (DATA(I),I=1,N)
0011          103 FORMAT(6F5.2)
0012              CALL CALC(MEAN,STD)
0013              WRITE(6,104) MEAN,STD
0014          104 FORMAT(' MEAN=',F8.4,' STD=',F8.4)
0015              GOTO 1
0016              END


0001              SUBROUTINE CALC(MEAN,STD)
0002              DIMENSION DATA(50)
0003              COMMON DATA,N
0004              REAL MEAN,MEAN2
0005              X = 0.0
0006              Y = 0.0
0007              DO 10 I=1,N
0008              X = X+DATA(I)
0009           10 Y = Y+DATA(I)*2
0010              MEAN = X/N
0011              MEAN2 = Y/N-MEAN**2
0012              STD = SQRT(MEAN2)
0013              RETURN
0014              END
```

Figure 1. Sample Program

Figure 2 gives the sample output from a sequence of commands used to debug the program. Input from the user is given in lowercase and output from SDS and the program is given in uppercase.

```
#debug mean
+READY
+run
  ENTER NUMBER OF DATA POINTS
   2
  ENTER DATA POINTS
   4.0  4.0

  SQRT ARGUMENT NEGATIVE
+CALL TO "MTS"
+READY
+set csect=calc
+break is#5 is#12
+DONE.
+run
  ENTER NUMBER OF DATA POINTS
   2
  ENTER DATA POINTS
   4.0  4.0
+AT BREAKPOINT IS#5 IN SECTION MAIN
+READY
+display n
+N IS NOT DEFINED IN THIS MODULE.
+set csect=*
+DONE.
+display n data(1) data(2)
+N  'F'  +2    (4 BYTES)
+DATA(1)  EL4'4.'
+DATA(2)  EL4'4.'
+continue
+AT BREAKPOINT IS#12 IN SECTION CALC
+READY
+display mean mean2
+MEAN DEFINITION USED FROM SECTION MAIN
+MEAN  EL4'0.25'
+MEAN2  EL4'-8.'
+set csect=calc
+DONE.
+display mean
+MEAN  EL4'4.'
+modify mean2 '0.0'
+MEAN2  EL4'-8.'
+NEW VALUE:  EL4'0.'
+continue
  MEAN=  4.0000 STD=  0.0
  ENTER NUMBER OF DATA POINTS
 $endfile
+USER PROGRAM RETURN
+READY
+stop
#
```

Figure 2. Sample Output

Since most users are incurable optimists when it comes to running a program for the first time, the RUN debug command is given to determine what the program will do on the first try. The comments "ENTER NUMBER OF DATA POINTS" and "ENTER DATA POINTS" are produced by the program, and therefore these two lines in the sample output do not start with the "+" prefix character. The program requires as a response an integer N of format I3 giving the number of data points to be used in the program. The input points are read into the array DATA which is of format 6F5.2.

A very simple set of test data is chosen for the first run. The size of the data set is 2 and consists of the points 4.0 and 4.0. This data set, using a simple mental calculation, will yield the results of 4.0 for the mean and 0.0 for the standard deviation. In choosing a test data set, it is wise to choose data which will give an obvious and simple answer so that any errors in the program will be readily apparent.

After the program is run, the comment "SQRT ARGUMENT NEGATIVE" appears, indicating that an erroneous call to the SQRT library subroutine was made in the CALC subroutine. The FORTRAN library has intercepted the call to SQRT and produced the message indicating that the value of the variable MEAN2 was negative. SDS intercepted the FORTRAN library's return to MTS and returned control to debug mode. Whenever any type of abnormal condition occurs during the execution of the program, such as a program interrupt or attention interrupt, SDS will step in and return control to debug command mode. This will also happen in the event of a call by the user's program to the system library subroutines SYSTEM, MTS, or ERROR.

At this point, if the user has a serially reusable program, he may rerun his program and monitor its performance more closely. For a program to be serially reusable, it must be capable of being rerun several times without being reloaded. All locations which contain constant values which are changed by the program must be initialized by the program <u>during</u> <u>execution</u>. For example, a program containing the statements

```
DATA I/3/
K = I
  .
  .
  .
I = 6
```

would not be reusable, since I would not be reinitialized to a value of 3; but a program containing

```
I = 3
K = I
  .
  .
```

```
   .
I = 6
```

would be reusable, since I is set to 3 each time the program is used. In general, serially reusable programs are easier to debug with SDS than are nonserially reusable programs, since they can be rerun several times without being reloaded. If the program were not serially reusable, then the user would have to reload the program again using the DEBUG command.

As an aid to monitoring the execution of the program, SDS provides the capability of setting breakpoints. When a breakpoint is encountered during execution of the program, execution is stopped, and control is returned to debug mode. The instruction at which the breakpoint is set has not yet been executed when execution is stopped.

The BREAK command may be used to set breakpoints by specifying the statement numbers at which execution is to be stopped. To refer to statement numbers in FORTRAN programs, a prefix must be used to distinguish the type of statement number being given. A "#" must prefix the statement number if it is an external (user-defined) statement number; e.g.,

```
BREAK #10
```

sets a breakpoint at the user-defined statement number 10. An "IS#" must prefix the statement number if it is an internal (source-listing) statement number; e.g.,

```
BREAK IS#10
```

sets a breakpoint at the source-listing statement 10. Only those statement numbers which define executable FORTRAN statements may be used. An executable statement is defined as a statement which is from one of the following categories:

(1) Assignment statements
(2) Control statements
(3) I/O statements

All others, such as those defining DIMENSION, REAL, INTEGER, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, and FORMAT statements will be undefined. Both internal and external statement numbers must be specified without leading zeros.

Since a program may consist of a main program and several subroutines and common sections, there must be a method for determining to which section statement numbers and other symbols refer. This may be done in two ways.

The SET CSECT command may be used to globally restrict all statement numbers and symbols to a specified section. In the sample output, the command sequence

```
SET CSECT=CALC
BREAK IS#5 IS#12
```

is used to set breakpoints at statements 5 and 12 of the subroutine CALC. If SET CSECT=CALC had not been given, then the first occurrence of IS#5 and IS#12 would be used. In this case, IS#12 would be in the section MAIN and IS#5 would be in the subroutine CALC since IS#5 is a FORMAT statement in MAIN. The command

```
SET CSECT=*
```

may be used to restore the searching of all sections. If the SET CSECT command has not been given, SDS searches all sections for statement numbers or variable names and use the first definition encountered.

The @C keyword modifier may be used to locally restrict a symbol to a specified section. The @C modifier applies only to the symbol to which it is appended and overrides any global restrictions set by the SET CSECT command. In the sample run, the command

```
BREAK IS#5 IS#12@C=CALC
```

also could have been used to set the breakpoints. The modifier @C=CALC restricts IS#12 to the subroutine CALC. @C=CALC is not needed for IS#5 since the only valid definition of IS#5 is in CALC.

The setting of breakpoints at the internal statements 5 and 12 of CALC was chosen so as to allow a closer inspection of the program near the area where the error was indicated. At statement 5, the input data may be examined before any actual calculations are made. At statement 12, the argument to the SQRT call may be examined.

After the breakpoints are set, the program is rerun. When the breakpoint at IS#5 is reached, execution is stopped and the message

```
AT BREAKPOINT IS#5 IN SECTION MAIN
```

is printed. At this point, the user may enter another debug command.

The DISPLAY command may be used to display variable locations in the program. Scalar variables are displayed by giving the variable name; e.g.,

```
DISPLAY MEAN
```

will display the contents of the variable MEAN converted according to its type and length. In this case, MEAN is a fullword real variable and its value is printed as

```
MEAN EL4'0.25'
```

The code EL4 indicates that the variable is real and four bytes in length. The codes for FORTRAN variables are:

E	Real (exponential or floating-point, 4-byte)
D	Real (floating-point, 8-byte)
F	Integer (fixed-point, 4-byte)
H	Integer (fixed-point, 2-byte)
L	Logical
M	Complex
X	Hexadecimal
I	Instruction

Array variables are displayed by giving the array name and its subscripts in the same manner as in the FORTRAN program; e.g.,

```
DISPLAY DATA(1)
```

will display the contents of the first element in the array DATA.

To display a variable which is in a blank-named common section, the @C modifier (or SET CSECT command) may be used with the name BLANK to specify the section. In the sample program, the array DATA is in the blank-named common section, hence

```
DISPLAY DATA(1)@C=BLANK
```

could have been used. Simply using

```
DISPLAY DATA(1)
```

would not have worked if the SET CSECT=* command had not been given first. Instead, an error message would be printed indicating that the symbol was undefined.

If all sections are open for searching, and if a symbol is used in more than one section (or subroutine), then SDS will display the first occurrence of that symbol and issue a warning message. In the example,

```
DISPLAY MEAN MEAN2
```

produced this message for MEAN since MEAN is defined in both the sections MAIN and CALC.

After the breakpoint at IS#5 has been reached, the next step is to display some of the input data values for the program to determine whether or not everything seems to be in reasonable order. The values of 2 for N and 4.0 for DATA(1) and DATA(2) indicate that the input data was correctly entered.

A CONTINUE command may then be given to resume execution of the program. After the breakpoint at IS#12 is reached, the user can again check the progress of the program. Displaying MEAN and MEAN2, it is discovered that the values are 4.0 and -8.0, respectively. A quick arithmetic check using the appropriate formulas

```
MEAN = (DATA(1)+DATA(2))/N
```

and

```
MEAN2 = (DATA(1)²+DATA(2)²)/N-MEAN²
```

yields the values 4.0 and 0.0, respectively. Hence, the value -8.0 is in error.

Looking back over the sample program, the user can see that this error was introduced in statement 9 of CALC. That statement should read

```
10 Y = Y+DATA(I)**2
```

Since it is not possible to recompile the program in SDS, the best that can be done at this point is to modify MEAN2 to contain the correct value. The MODIFY command may be used to do this. The first parameter for this command gives the name of the variable to be modified. The second parameter gives the value to be used in the modification; the value must be enclosed in primes, e.g.,

```
MODIFY MEAN2 '0.0'
```

The value for MEAN2 is now modified to 0.0, and execution of the program may be resumed to determine if the remainder of the program seems to be correct. This time, the correct values for the test data are printed by the program.

Instead of entering a second set of test data, the user will probably want to recompile the program to correct the error in CALC. To terminate the program, the user enters a $ENDFILE (or equivalent). SDS intercepts the termination of the program and returns control to debug mode. The STOP command may be then used to return control to MTS.

The user may use the RESTORE and CLEAN commands to remove breakpoints from the program that were set by the BREAK command. The RESTORE command will remove a specified breakpoint; e.g.,

```
RESTORE IS#12
```

will remove the breakpoint set at statement 12 in CALC. The CLEAN command will remove all breakpoints that are set in the program.

Multidimensioned arrays are specified in the same manner as linear arrays. For example, the third element in the array specified by the FORTRAN source statement

```
DIMENSION ALPHA(10,10)
```

may be displayed by

```
DISPLAY ALPHA(3,1)
```

A sequence of elements of an array may be displayed using the block notation format. For example, to display the first ten elements of ALPHA, the user may specify

```
DISPLAY ALPHA(1,1)...(10,1)
```

The user should note that in FORTRAN programs, arrays are stored in ascending locations with the first subscript increasing the most rapidly and the last subscript the least rapidly.

Arrays may also be displayed using symbolic subscripts. If, in the FORTRAN program, the variables I and J have the values 2 and 3, respectively, then

```
DISPLAY ALPHA(I,J)
```

will display the element ALPHA(2,3).

Arguments to FORTRAN subroutines and functions may be one of two types:

(1) reference by value, or
(2) reference by location.

When an argument is passed as a reference by value argument, the actual value of the variable is passed by the calling program to the subprogram. Therefore, there is a copy of that variable in both the calling program and the subprogram. Scalar (undimensioned) arguments are normally passed in this manner. The subprogram uses its own copy of the argument for any calculations done. Upon return of the subprogram to the calling program, the argument is passed back to the calling program and the calling program's copy is updated. Therefore, when displaying an argument of this type, it is important to keep in mind where the variable is located and when it is displayed.

When an argument is passed as a reference by location argument, only the address of the argument is passed by the calling program to the subprogram. Therefore, only one copy of the argument exists and it is located in the calling program (or a common section). Array arguments are always passed in this manner. The subprogram uses the copy of the argument in the calling program for its calculations. When displaying an argument of this type, either the variable name from the calling program or the variable name from the subprogram argument list may be used. Both refer to the same variable. When using the name from the subprogram argument list, the address passed to the subprogram is used to locate the variable in the calling program. Therefore, the subpro-

gram must have been called at least once for this address to be valid. If the address is invalid, an error comment is produced in form

```
xxxxxxx SPECIFIES AN ILLEGAL ADDRESS.
```

Most debug commands may be given in an abbreviated format. The minimum abbreviations that may be used are underlined.

BREAK	RESTORE
CLEAN	RUN
CONTINUE	SET CSECT
DISPLAY	STOP
MODIFY	

An automatic error-dumping facility similar to that provided by the MTS SET ERRORDUMP command is provided for batch users. In the event of an error condition occurring during the execution of the program, a symbolic dump will be given of the program. This dump will include all variable locations in the program. This facility may be activated for the sample program by the command sequence

```
$SET DEBUG=ON
$SDS SET ERRORDUMP=ON
$RUN MEAN
  2
  4.0  4.0
$ENDFILE
```

Note that the MTS RUN command has been given instead of the DEBUG command. The error-dump facility may be deactivated by the command

```
$SET DEBUG=OFF
```

The symbolic dump will give the variable storage for the sample program in a format similar to the following:

```
DUMP OF SECTION             VA=5004F0

  RA     SYMBOL    TYPE    VALUE                          HEX VALUE

000000  DATA(1)    'E'  4.0000000                        41400000
000004  DATA(2)    'E'  4.0000000                        41400000
000008  DATA(3)    'E'  0.0E+00                          81818181
 ...      ...              ...
0000C4  DATA(50)   'E'  0.0E+00                          81818181
0000C8  N          'F'  +2                               00000002


DUMP OF SECTION MAIN        VA=5002A8

  RA     SYMBOL    TYPE    VALUE                          HEX VALUE

0000B0  I          'F'  +2                               00000002
0000B4  MEAN       'E'  0.0E+00                          81818181
0000B8  STD        'E'  0.0E+00                          81818181


DUMP OF SECTION CALC        VA=5005C0

  RA     SYMBOL    TYPE    VALUE                          HEX VALUE

0000A0  X          'E'  8.0000000                        41800000
0000A4  Y          'E'  16.000000                        42100000
0000A8  I          'F'  +2                               00000002
0000AC  MEAN       'E'  4.0000000                        41400000
0000B0  MEAN2      'E'  -8.0000000                       C1800000
0000B4  STD        'E'  0.0E+00                          81818181
```

FORTRAN-H and VS FORTRAN programs may also be debugged using SDS. To generate a FORTRAN-H object module with SYM records, the FORTRAN-H compiler should be invoked with the TEST option using a command of the form:

```
$RUN *FTN SCARDS=source SPUNCH=object PAR=OPT=H,TEST,options
```

To generate a VS FORTRAN object module with SYM records, the VS FORTRAN compiler should be invoked with the SYM option using a command of the form:

```
$RUN *FORTRANVS SCARDS=source SPUNCH=object PAR=SYM,options
```

Because of the optimizing features of the FORTRAN-H and VS FORTRAN compilers, the symbol table information provided may be of limited use. This is due to several possible transformations which may be performed on the object module by the compiler during optimization. For example, the compiler often moves operations from within a statement to the beginning of the block of statements in which that statement resides if it does not affect the logical operation of the program. This makes it quite difficult to follow the exact execution flow of the program using SDS. For instance, a breakpoint may be set at a statement label, and when the breakpoint is reached, the statement may have already been executed because its text has been moved to the beginning of the block. Furthermore, it is not uncommon for the compiler to move the entire text of a statement, leaving only the label. When this happens to two adjacent labeled statements, both labels reference the same location and SDS does not treat them as distinct (since, in fact, they are not). Another difficulty posed by optimization is the fact that variable values are often kept in registers for large ranges of instructions without updating the memory location. This often happens within DO-loops. Since SDS knows only which memory location corresponds to which symbol (and not which register), displaying a variable from SDS may not yield the current value of the variable.

The problems of optimization are eliminated if the program is compiled at optimization level 0. Unfortunately, many of the advantages of using FORTRAN-H or VS FORTRAN are also eliminated at optimization level 0.

The FORTRAN-H compiler only produces symbol table information for external (user-defined) statement labels; no information is produced for internal (source-listing) statement labels. As with FORTRAN-G, the label is preceded by "#", e.g., statement label 100 is the symbol #100.

INDEX

Reader's Comment Form

FORTRAN in MTS
Volume 6
October 1983
(February 1988 Reprint)

Errors noted in publication:

Suggestions for improvement:

Your comments will be much appreciated. Send the completed form to the Computing Center by Campus Mail or U.S. Mail, or drop it in the Suggestion Box at the Computing Center, NUBS, or UNYN.

Date _____

Name _____

Address _____

_____

_____

Publications
Computing Center
University of Michigan
Ann Arbor, Michigan 48109

Update Request Form

FORTRAN in MTS
Volume 6
October 1983
(February 1988 Reprint)

Updates to this manual will be issued periodically as errors are noted or as changes are made to MTS. If you would like to have these updates mailed to you, please submit this form.

Updates are also available in the files at some of the Computing Center's larger public stations such as NUBS and UNYN; there you may obtain any updates to this volume that may have been issued before the Computing Center receives your form. Please indicate below if you want to have the Computing Center mail you any previously issued updates.

Name ———————————————————————————

Address ———————————————————————————

———————————————————————————

———————————————————————————

Previous updates needed (if applicable):———————

Send the completed form to the Computing Center by Campus Mail or U.S. Mail, or drop it in the Suggestion Box at the Computing Center, NUBS, or UNYN. Local users should give a Campus Mail address.

Publications
Computing Center
The University of Michigan
Ann Arbor, Michigan 48109

Users associated with other MTS installations (except the University of British Columbia) should return this form to their respective installations. Addresses are given on the reverse side.

Addresses of other MTS installations:

Publications Clerk
352 General Services Bldg.
Computing Services
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Information Officer, NUMAC
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne
England NE1 7RU

Rensselaer Polytechnic Institute
Documentation Librarian
310 Voorhees Computing Center
Troy, New York 12181

Simon Fraser University
Computing Centre
User Services Information Group
Burnaby, British Columbia
Canada V5A 1S6

Wayne State University
Computing Services Center
Academic Services Documentation Librarian
5925 Woodward Ave.
Detroit, Michigan 48202