SC28-6891-1

Program Product

IBM VM/370 (CMS) Terminal User's Guide for FORTRAN IV Program Products

Program Numbers 5734-FO1
5734-FO2
5734-FO3
5734-LM1
5734-LM3

IBM

SC28-6891-1

Program Product

IBM VM/370 (CMS) Terminal User's Guide for FORTRAN IV Program Products

Program Numbers 5734-FO1
5734-FO2
5734-FO3
5734-LM1
5734-LM3

IBM

Second Edition (April 1975)

This edition, as amended by technical newsletters SN20-9201 and SN20-9225, applies to Release 1.0 of the IBM Virtual Machine Facility/370 (VM/370) (CMS).

This edition is a reprint of SC28-6891-0 incorporating changes released in Technical Newsletters SN28-0609 (dated March 1, 1973) and SN28-0620 (dated January 3, 1974). Changes are listed in the Summary of Amendments, Number 3, on the facing page.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using the publication, consult the latest *IBM System/360 Bibliography*, GC20-0360, or *IBM System/370 Bibliography*, GC20-0001, and the technical newsletters that amend the particular bibliography, to learn which editions are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office that serves your locality.

Forms for readers' comments are provided at the back of this publication. If the forms have been removed, address comments to IBM Corporation, P. O. Box 50020, Programming Publishing, San Jose, California 95150. Comments and suggestions become the property of IBM.

Summary of Amendments Number 1

Date of Publication: March 1, 1973

Form of Publication: TNL SN28-0609 to SC28-6891-0

CP and CMS Command Abbreviations

Maintenance: Documentation Only

Valid abbreviations have been added to the summary descriptions of significant CP and CMS commands.

XTENT Option

New: Documentation Only

A description of the XTENT option of the FILEDEF command has been added for users of direct access files.

Flagging of Data Spill

New: Documentation Only

A statement has been added to the description of data spill indicating that several compilers will flag spill as an error even though they process it correctly.

Reproduction of Command Formats for Internal Use Only

New: Documentation Only

Footnotes have been added to the sections describing the compiler command formats. These footnotes indicate that users may copy the sections for internal use only.

H Extended SIZE Option

New: Documentation Only

The description of the SIZE option for the H Extended compiler has been expanded to reflect the operation of the option and to guide the user in its use.

Asynchronous I/O Message

New: Programming and Documentation

A message indicating that an asynchronous I/O operation has been attempted has been added to the restriction on asynchronous I/O.

Terminal Listing Sheet Examples

Maintenance: Documentation Only

Examples showing terminal listing sheets have been revised to more accurately reflect their actual appearance.

Foldout Pages for the Sample Terminal Session

Maintenance: Documentation Only

The terminal listing sheets for the sample terminal session have been printed on foldout pages for ease of reference.

Editorial changes having no technical significance are not noted here.

Specific changes to the text as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 2

Date of Publication: January 3, 1974

Form of Publication: TNL SN28-0620 to SC28-6891-0 as amended by TNL SN28-0609

CMS Support of FREEFORM Source Programs

New: Programming and Documentation

A description of preparing free form source programs and of the new filetype FREEFORT has been added.

SIFT Utility

New: Programming and Documentation

A description of the changes made to the SIFT Utility program in support of free-form source files has been added.

ASA Carriage Control Characters

Modification: Documentation Only

The character + has been removed from the list of supported ASA carriage control characters.

OS File Compatability

Modification: Documentation Only

Restrictions have been added to the description of file compatability and conditions under which it can be accomplished are outlined.

Editorial changes having no technical significance are not noted here.

Specific changes to the text of this publication are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 1

Date of Publication: March 1, 1973

Form of Publication: TNL SN28-0609 to SC28-6891-0

CP and CMS Command Abbreviations

Maintenance: Documentation Only

Valid abbreviations have been added to the summary descriptions of significant CP and CMS commands.

XTENT Option

New: Documentation Only

A description of the XTENT option of the FILEDEF command has been added for users of direct access files.

Flagging of Data Spill

New: Documentation Only

A statement has been added to the description of data spill indicating that several compilers will flag spill as an error even though they process it correctly.

Reproduction of Command Formats for Internal Use Only

New: Documentation Only

Footnotes have been added to the sections describing the compiler command formats. These footnotes indicate that users may copy the sections for internal use only.

H Extended SIZE Option

New: Documentation Only

The description of the SIZE option for the H Extended compiler has been expanded to reflect the operation of the option and to guide the user in its use.

Asynchronous I/O Message

New: Programming and Documentation

A message indicating that an asynchronous I/O operation has been attempted has been added to the restriction on asynchronous I/O.

Terminal Listing Sheet Examples

Maintenance: Documentation Only

Examples showing terminal listing sheets have been revised to more accurately reflect their actual appearance.

Foldout Pages for the Sample Terminal Session

Maintenance: Documentation Only

The terminal listing sheets for the sample terminal session have been printed on foldout pages for ease of reference.

Editorial changes having no technical significance are not noted here.

Specific changes to the text as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Form of Publication: TNL SN20-9201 to SC28-6891-1

Default Record Format, Logical Record Length, and Block Size

Maintenance: Documentation Only

The default record format, logical record length, and block size when using the FILEDEF command has been added.

Listing Produced When the PRINT Option is Used

Maintenance: Documentation Only

If the FORTHX command is entered with the PRINT option, a listing is produced at the offline printer instead of the primary disk.

Base Register Usage When Using the FORTRAN IV (H Extended) Compiler

Maintenance: Documentation Only

A description of base register usage in an object program compiled by the FORTRAN IV (H Extended) compiler has been added.

Registers Reserved for Branch Optimization

Maintenance: Documentation Only

A description of the registers reserved for branch optimization has been added.

Miscellaneous:

Maintenance: Documentation Only

Various examples have been corrected and/or expanded.

Editorial changes having no technical significance are not noted here.

Specific changes to the text as of this publishing date are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Summary of Amendments — Number 2

Date of Publication: January 3, 1974

Form of Publication: TNL SN28-0620 to SC28-6891-0 as amended by TNL SN28-0609

CMS Support of FREEFORM Source Programs

New: Programming and Documentation

A description of preparing free form source programs and of the new filetype FREEFORT has been added.

SIFT Utility

New: Programming and Documentation

A description of the changes made to the SIFT Utility program in support of free-form source files has been added.

ASA Carriage Control Characters

Modification: Documentation Only

The character + has been removed from the list of supported ASA carriage control characters.

OS File Compatability

Modification: Documentation Only

Restrictions have been added to the description of file compatability and conditions under which it can be accomplished are outlined.

Editorial changes having no technical significance are not noted here.

Specific changes to the text of this publication are indicated by a vertical bar to the left of the text. These bars will be deleted at any subsequent republication of the page affected.

Preface

This user's guide is intended for FORTRAN programmers who will be using the IBM System/360 OS FORTRAN IV (G1) or Code and Go FORTRAN IV compiler and the FORTRAN IV Library (Mod I) or the FORTRAN IV (H Extended) compiler and the IBM FORTRAN IV Library (Mod II) under the control of the Conversational Monitor System component of the Virtual Machine Facility/370. It is assumed that the reader is familiar with the FORTRAN IV Language and the CMS component of VM/370.

This publication is divided into 9 parts as follows:

- Introduction
- What You Need To Know before Using CMS and the FORTRAN IV Compilers for the First Time
- Sample CMS Terminal Session
- VM/370 Commands for the FORTRAN IV Programmer
- CMS Programming Considerations
- FORTRAN IV Programming Considerations
- Using the FORTRAN IV Compilers
- Loading and Executing FORTRAN TEXT files under CMS
- Appendixes

The "Introduction" briefly describes the operation of CMS and the relation of the FORTRAN IV compilers and libraries to that system.

The part "What You Need To Know before Using CMS and the FORTRAN IV Compilers for the First Time" lists information about CMS and the compilers that a new programmer must obtain from the system administrator in his computing center before using the system.

The "Sample CMS Terminal Session" illustrates a typical terminal session and introduces a less experienced user to some of the commands and techniques necessary to write, compile, and execute a FORTRAN program under CMS.

The "VM/370 Commands for the FORTRAN IV Programmer" part lists the system commands that the FORTRAN programmer typically needs or uses. The list does not include all the VM/370 commands available.

The "CMS Programming Considerations" part describes general concepts in CMS file management and definition for the FORTRAN programmer. It describes the creation of source files, the characteristics of compiler output files, and the creation and use of files during the execution of object programs.

The "FORTRAN IV Programming Considerations" part describes FORTRAN IV language coding techniques that will make most efficient use of the compilers that you will be using under CMS. In addition, it describes the use of the FORTRAN IV libraries and special features that are available to particular compilers.

The part "Using the FORTRAN IV Compilers" provides specific information on the use of the FORTRAN IV (G1), Code and Go FORTRAN IV, and FORTRAN IV (H Extended) compilers by the CMS terminal user. It describes the commands necessary to invoke these compilers and the various kinds of output that are available. In addition, it describes the restrictions placed upon the FORTRAN IV language by the compilers. Each compiler is treated separately for ease of reference.

The part "Loading and Executing FORTRAN TEXT files under CMS" presents the commands necessary to load and execute FORTRAN programs. The commands required for each compiler are treated separately.

The "Appendixes" contain a description of the error messages produced by CMS for the FORTRAN programmer, information on using assembler language subprograms with FORTRAN programs, a description of the FORTRAN IV Debug Facility, a description of the CONVERT utility program that is available to Code and Go FORTRAN IV programmers who want to convert free-for source programs to fixed-form, information on modifying the extended error handling facility option table, and a description of file characteristics for compatability with OS data sets.

Industry Standards Reflected in this Product

This product is designed according to the specifications of the American National Standard (ANS) FORTRAN, X3.9-1966, as understood and interpreted by IBM as of December 1972.

Reference Publications

Information on the IBM FORTRAN IV Language and the IBM FORTRAN IV Libraries (Mod I) and (Mod II) can be found in the following publications:

IBM System/360 and System/370
FORTRAN IV Language
Order No. GC28-6515

IBM System/360
FORTRAN IV Library
Mathematical and Service Subprograms
Order No. GC28-6816

IBM System/360 OS
FORTRAN IV Mathematical and Service Subprograms
Supplement for the Mod I and Mod II Libraries
Order No. SC28-6864

Diagnostic messages for the FORTRAN IV (G1) compiler can be found in the following publication:

IBM FORTRAN IV (G1) Processor
and TSO FORTRAN Prompter
for OS and VM/370 (CMS)

Installation Reference Material
Order No. SC28-6856

Diagnostic messages for the Code and Go FORTRAN IV compiler can be found in the following publication:

IBM Code and Go FORTRAN IV Processor
for OS and VM/370 (CMS)
Installation Reference Material
Order No. SC28-6859

Diagnostic messages for the FORTRAN IV Library (Mod I) can be found in the following publication:

IBM FORTRAN IV Library (Mod I)
for OS and VM/370 (CMS)
Installation Reference Material
Order No. SC28-6858

Diagnostic messages for the FORTRAN IV (H Extended) compiler and (Mod II) library can be found in the following publication:

IBM System/360 OS
FORTRAN IV (H Extended) Compiler
and Library (Mod II)
Messages
Order No. SC28-6865

Information on VM/370 and the command langauge can be found in the following publications:

IBM Virtual Machine Facility/370
Command Language
User's Guide
Order No. GC20-1804

IBM Virtual Machine Facility/370
EDIT Guide
Order No. GC20-1805

IBM Virtual Machine Facility/370
Terminal User's Guide
Order No. GC20-1810

Information on using the FORTRAN IV compilers under OS can be found in the following publications:

IBM System/360 OS
Code and Go FORTRAN and
FORTRAN IV (G1)
Programmer's Guide
Order No. SC28-6853

IBM System/360 OS
FORTRAN IV (H Extended) Compiler
Programmer's Guide
Order No. SC28-6852

Contents

Figures

Introduction

As a FORTRAN programmer your are probably familiar with *batch processing,* punching a card deck, sending it to your computing center, and waiting several hours or overnight to get your results back. This arrangement works well for large production programs that are run on a recurring basis, for programs that handle large amounts of card or tape files and produce voluminous printed listings, and for programs in which time is not a vital factor. However, for problem solving, quick retrieval of information, or system maintenance and modification, batch processing does not always fill your needs. Often, for example, a small error in an urgently needed program will keep it from running successfully and the time you spent waiting and rerunning the program was wasted. When time is critical, a different type of processing is necessary.

A *time-sharing* system is the answer. It allows you to sit at a terminal, use simple commands, enter your program, run it, and get your results back in a matter of minutes at the same terminal. The Conversational Monitor System (CMS), operating in the time-sharing environment produced by the Control Program (CP) component of the Virtual Machine Facility/370, offers you, the terminal user, an extensive range of computer functions: console control, creating and managing files, compiling and executing programs, performing input and output operations, and system development and maintenance.

The control program component of VM/370 creates a simulated (that is, *virtual*) computer and makes it available on a shared-time basis. Each user has his own simulated computer and, shares with you, the time and facilities of the real computer in your computing center. Your terminal becomes the operator's console for your virtual computer. The CP command language permits you to control the operation and status of your virtual computer in the same way that a computer operator controls the real machine. With some of the commands available, you can initialize control programs, manipulate devices and data, and communicate with other users or with the system operator, who runs the real computer to which your terminal is attached.

As a real computer functions most efficiently under the control of an operating system, so too does a virtual machine. The Conversational Monitor System is an operating system that you can execute under CP to control your virtual computer. CMS permits you to use your terminal as the primary means for entering data and writing programs. The CMS command language simplifies file and data handling through the CMS editor and its various subcommands, and minimizes your concern with system functions and elaborate data management procedures. CMS is, primarily, disk oriented. Most files are kept on disks and are always available to you through your terminal. CMS allows you to create files that contain virtual card decks, printed listings, and magnetic tapes. This means that you can, for example, create a virtual card deck at your simulated card punch, use CMS commands to transfer it to your simulated card reader, and read it back in without having to handle actual cards. Of course, you can read actual cards and tapes, and create real card decks, printed listings, and tape files off-line on real devices in your computing center or a remote entry system. In addition, you may group a series of related or often used CMS commands together and execute them as a unit, thus simplifying your use of the command language.

With the addition of the FORTRAN compiler commands to the CMS command language, you can compile FORTRAN source programs and use the FORTRAN library subprograms. The TEXT files created by the FORTRAN compilers under CMS can be loaded and executed under CMS or link edited and executed under OS. In addition object programs created under OS can be loaded and run under CMS. The FORTRAN compilers and library that are available as program products under CMS are:

- FORTRAN IV (G1) Compiler
- Code and Go FORTRAN IV Compiler
- FORTRAN IV (H Extended) Compiler
- FORTRAN IV Library (Mod I) (for G1 and Code and Go)
- FORTRAN IV Library (Mod II) (for H Extended)

FORTRAN IV (G1) Compiler

The FORTRAN IV (G1) compiler is invoked under CMS with the FORTGI command. FORTRAN IV (G1), offers the capabilities of directing error diagnostics and/or compiler output to a terminal and of using list-directed input/output. Additionally, the processor supports FORTRAN Interactive Debug.

Code and Go FORTRAN IV Compiler

The Code and Go FORTRAN IV compiler is invoked under CMS with the GOFORT command. Code and Go FORTRAN, as a time-sharing tool, has been designed to meet the specific needs of two types of users: (1) the *problem solving* programmer, who writes, debugs, and executes relatively short programs at the terminal, and (2) the *production* programmer who debugs components of a large program on-line before running the program through a production-oriented processor, such as FORTRAN IV (H Extended). Thus, design emphasis has been placed on rapid compilation-execution turnaround and on ease of use. Code and Go supports free-form input format -- which considerably reduces the programmer's concern with terminal-typing tasks, such as tab settings and margin stops -- and includes options for obtaining short- or long-form diagnostic messages. Support is also provided for FORTRAN Interactive Debug and for the use of list-directed input/output, which frees the programmer from having to code FORMAT statements.

FORTRAN IV (H Extended) Compiler

The FORTRAN IV (H Extended) compiler is invoked under CMS with the FORTHX command. FORTRAN IV (H Extended), besides providing extended language capability for computational power, is a true production compiler, utilizing advanced optimization technology to produce efficient object code. The extended language capabilities of FORTRAN IV (H Extended) include:

- Support for extended precision arithmetic via REAL*16 and COMPLEX*32 data types or via use of a compiler option.
- Automatic function selection to simplify references to built-in and library functions.

As a compilation-time option, the user may specify automatic precision increase, allowing for conversion of floating-point calculations from single to double and double to extended precision. The FORTRAN IV (H Extended) compiler requires the FORTRAN IV Library (Mod II), or its equivalent, for compiling and executing source programs.

FORTRAN Library (Mod I)

The FORTRAN IV (Mod I) library is made available for use under CMS with the GLOBAL TXTLIB command specifying from one to eight installation designated library names. Code and Go and G1 are supported by the FORTRAN IV Library (Mod I), which provides mathematical, service, and input/output routines needed by the processors. Additionally, the Mod I library and the processors incorporate the same data conversion routines, which round real constants and real data items on input rather than truncate them. This provides finer resolution and greater accuracy of results.

FORTRAN IV Library (Mod II)

The FORTRAN IV (Mod II) library is made available for use under CMS with the GLOBAL TXTLIB command specifying from one to eight installation designated names. FORTRAN IV (H Extended) compiler is supported by the FORTRAN IV Library (Mod II), in addition to specifically providing routines required by the processor, encompasses all of the functions of the Mod I library; thus, an installation equipped with the Mod II library does not need the Mod I library to support Code and Go or G1.

What You Need To Know before Using CMS or the FORTRAN IV Compilers for the First Time

Information About CMS

Before you use CMS or the FORTRAN IV compilers for the first time, you should obtain the following information from the system administrator in your computing center:

For Gaining Access to the System

- What is your user identification code?

  These are unique names that identify you and authorize your use of VM/370.

- What is your log-in procedure?

  The log-in procedure may vary depending on the type of terminal that you will be using and the way it is connected to the real computer in your computing center. This information can also be found in the publication *IBM VM/370 Terminal User's Guide,* Order No. GC20-1810.

For Your Terminal

- What is the character-delete character?

  You must determine which character you will use for deleting a character from a line.

- What is the line-delete character?

  You must determine which character you will use for deleting an entire line.

- What is the line-end character?

  You must determine which character you will use for logically ending an input line.

  Note: This information is also available through the QUERY TERMINAL command.

For Your Virtual Machine

- What is the configuration of your virtual machine?

  The needs of your work and the type of programs you will be using should help your system administrator decide on the best configuration for your virtual machine.

- What disk is normally assigned as your primary (A) disk?

  Your primary disk is normally assigned to you at the beginning of each terminal session. It is usually identified as 191. Your system may require a different disk and you may have to issue an ACCESS command for it at the start of your session.

For Your FORTRAN Compiler

- Which FORTRAN IV compilers and libraries are available?

  You will need to know which FORTRAN IV compilers, libraries, and optional features are available in order to select the compiler that is best suited to your needs.

- What defaults have been established for the compiler and library you are going to use?

  You should determine what defaults have been established for: the compiler options, the maximum number of FORTRAN data set reference numbers permitted, the data set reference numbers that have been pre-defined for the READER, PRINTER, and PUNCH files, and the names of the files in which your library is available. Some of the defaults in use at your computing center may differ from those described in this book and may require different techniques than those described. The defaults in use may differ, since your system administrator has the ability to tailor his system to better meet the needs of his users.

Syntax Conventions Used in This Book

The syntax conventions used to illustrate CMS commands and FORTRAN statements throughout this book are:

- Lower-case letters, digits, and special characters represent information that you must type exactly as shown.
- Upper-case letters digits and special characters represent information that is typed out by the system.
- Italics represent information that you must supply.
- Information contained within brackets [] is optional and may be omitted. Where a list is given any number of the items listed may be included.
- The appearance of braces {} indicates that a choice must be made between the items contained in the braces.
- The appearance of the vertical bar | indicates that a choice must be made between the item to the left of the bar and the item to the right of the bar.
- An ellipsis (a series of three periods) indicates that the preceding syntactical unit may be used one or more times in succession.
- A list whose length is variable is specified by the format: $x_1,x_2,...,x_n$. This format indicates that a variable number of items may be specified, but that at least one is required (commas must separate the items).

Sample Terminal Session

This section describes a hypothetical terminal session using the FORTRAN IV (G1) compiler; however, any of the other FORTRAN compilers that are available could be used. You will be introduced to many of the features of VM/370 and CMS and how they interact with you. You are invited to sit at your terminal and work along as the sample session is described. If you follow the session closely, you will begin to develop a facility for programming at a terminal under the control of CMS. This sample session does not illustrate all the commands that are available to you, only those that are directly related to creating, compiling, and executing a FORTRAN program. An explanation of all the commands is contained in the publication *IBM VM/370 Command Language User's Guide,* Order No. GC20-1804. It is assumed throughout that you are using an IBM 2741 Communications Terminal; if you are not, refer to the publication *IBM VM/370 Terminal User's Guide,* Order No. GC20-1810 for information on your terminal.

As you work through the session, remember that with CMS there is usually more than one way to achieve a desired result. The techniques and commands outlined here may not be the ones that you will eventually decide to use; however, they serve as a guide and represent a usable programming tool.

A printout of the sample session, as it would appear at your terminal, is included on foldout charts that follow the descriptive text. Turn to the first chart now and keep it open as you read about the sample session. The circled numbers in the left-hand margin of the printout indicate important points in the session. Each number has a corresponding explanation in the text. In this session, all of the commands and codes that you will type are printed in lowercase letters and all the systems responses are printed in uppercase letters (this convention is followed for all the illustrations in this book).

Preliminary Procedures and Signing On

1 The first thing you must do to start a terminal session is to turn on your terminal according to the instructions provided by your installation. In many cases an instruction sheet, such as the one shown in Figure 1, will be attached to your terminal. In the example shown, steps 1 through 8 must be done to turn on the power and establish a connection with the system. The meaning of step 9 will become evident in the description of the sample session that follows. If an instruction sheet is not available at your terminal, consult the publication *IBM VM/370 Terminal User's Guide,* Order No. GC20-1810 or your computing center.

Terminal #7

(Available from 9:00 a.m. - 3:00 p.m. For additional time call E. Souse at extension 7801)

1. Turn the ON/OFF switch to ON.
2. Make sure that the COM/LCL switch is set to COM.
3. Remove the handset from the attached telephone (data set).
4. Press the TALK button on the telephone.
5. Dial extensions 5555, 5556, or 5557.
6. Wait for the high-pitched tone. When you hear this tone, you are in contact with the computer. If you get a busy signal or no answer, hang up and repeat this procedure starting from step (3) trying another extension.
7. Upon hearing the high-pitched tone, push the DATA button on the telephone. If the DATA button light goes off at any point during the session, repeat this procedure from step (3).
8. Replace the handset on the cradle.
9. Enter the LOGIN command.

Note: When you are finished with your terminal session, enter the LOGOFF command, and turn the ON/OFF switch to OFF. The DATA button light will go out.

Figure 1. Sample Instruction Sheet for a Hypothetical Terminal

Starting the Session

2 The first entry on the printout of the sample session is the system's response to your call, notifying you that you have been successfully connected to the system and that VM/370 is available. The format of the VM/370 ONLINE response varies slightly depending upon the type of terminal you are using. (See the publication *IBM VM/370 Terminal User's Guide,* Order No. GC20-1810 for the exact response that your terminal types out.) Once you have received the ONLINE response, you are ready to identify yourself to the system.

3 To do so, hit the ATTN key. When the keyboard unlocks, type a LOGIN command. The system recognizes authorized users by an identifier and a password that are entered separately. As you can see, you begin by typing your identifier (here, EUSTACE) as part of the LOGIN command. The system will then ask for your password. Depending on your terminal, the password either will not be printed or will be typed over and obscured. This is a safeguard to protect your user identification from unauthorized use. If the system recognizes you as an authorized user, it will type out a LOGIN message and any messages from the system operator or other users. The LOGIN

message indicates that your virtual computer is available and you may now initialize an operating system in it.

4 Initializing an operating system, in this case CMS, is simple: type the abbreviation IPL followed by at least one blank and CMS. CMS responds, indicating that it has been successfully initialized. At this point you are no longer directly connected to the control program component of VM/370, but are now in communication with CMS. All the CMS commands are now available to you. Should you wish to return to the control program environment you need only hit the ATTN key twice or type the letters CP. To go back to CMS again, type BEGIN. There is a mechanism that provides an easy check to determine which component you are communicating with. Simply enter a null line (that is, a line containing no characters or blanks) by hitting the RETURN key CR (representing carrier return). The system will identify the component you are communicating with by typing the message CP or CMS.

It is assumed for this terminal session, that you already have a PROFILE EXEC procedure available that contains a GLOBAL TXTLIB command specifying the entire FORTRAN IV Library that you are going to use. If you do not have such a procedure, it is advisable to create one, now, before you continue on. A PROFILE EXEC procedure that has been filed on your A disk is executed automatically when you issue the first command. The procedure usually contains CP and CMS commands that you would need each time you initialized CMS and that would be required for the type of work you are going to do. For example, a typical PROFILE EXEC procedure might contain the following:

ACCESS commands - to obtain disks other than A191

SET commands - to establish a terminal environment

GLOBAL commands - to make text and macro libraries available

To create a PROFILE EXEC procedure now, use the EDIT facility enter the commands that you want the procedure to contain, and reinitialize CMS. See the publication *IBM VM/370 Command Language User's Guide,* Order No. GC20-1804 and *IBM VM/370 EDIT Guide,* Order No. GC20-1805. For detailed information on creating an EXEC procedure. Remember, if you do not have a PROFILE EXEC procedure on your A disk, you must issue any of the required commands listed above at the beginning of each terminal session. To use this terminal session you will require one GLOBAL TXTLIB command specifying the entire FORTRAN IV library you are going to use.

5 Now that you have a computer and an operating system at your disposal, you are ready to begin work. Since you will not be using punched cards, and all your data will be kept in the system's internal storage, you must identify your collection of data, called files, to the system (your collection of data in this case is a program). To do so, you need only choose a unique name (one that does not already exist on any of your disks) for it. Strictly speaking, a CMS file is properly identified by filename, filetype, and filemode. The filename identifies the file; the filetype indicates the contents of the file, and the filemode determines on which of your disks the file is to be placed. Only the filename and filetype are required for the purposes of this example.

The default filemode of A1 is accepted throughout. To avoid the use of duplicate filenames, type a LISTFILE command. CMS will type a list of all the files that are being kept on your disks.

6 You will notice that the system types the characters R; after the list of files. This is a ready message that indicates the system has successfully completed your request and is ready for the next command. Should the system fail to operate properly after the message was typed out, all of the information preceeding it will be available. This means that if you are in the middle of a long and complicated series of commands and code, you do not have to start over should the system fail; you can begin, after reinitializing the system, immediately following the last ready message.

7 After choosing a name for your file, in this case MAGICSQ, you can use the EDIT command to identify it to the system. By entering the filename (and, since this will be a FORTRAN source program, a filetype of FORTRAN,) the new file will be created under the filename of MAGICSQ. The filetype, FORTRAN, tells the CMS editor that your program statements are to be recorded in a FORTRAN format. (There are other filetypes that you will encounter later in this book.) Since this is a new file, the CMS editor responds with NEW FILE and enters the EDIT mode. In the EDIT mode, all the editor subcommands are now available to you. Before you can begin writing your program, however, you must issue an INPUT subcommand. This command causes the editor to enter the INPUT mode, in which all subsequent lines will be treated as input data and will become part of the file MAGICSQ. Since the editor recognizes the FORTRAN filetype, an internal set of tabs is established. By depressing the TAB key editor will automatically begin entering the line in column 7; thereby, saving you the trouble of spacing to column 7.

8 You are now ready to begin writing your FORTRAN program. The program that you will write generates a magic square for a number that you provide from the terminal. The magic square for a number consists of a set of numbers (none of them repeated) that are arranged into a square, so that each row, column, and diagnonal add up to the original number.

Example:

This is the magic square for the number 45.

```
18  11  16

13  15  17

14  19  12
```

To keep this program relatively simple, the magic square produced is limited to 3-by-3 and the numbers, for which the square can be generated, must be greater than 15 and divisible by 3. In accordance with good programming practice comments have been included that describe what the program does and how it works. Additional comment lines, containing the letter C, have been included to highlight the text of the comments and make them easier to read. The program

has been purposely written with errors built-in to demonstrate some of the facilities of the CMS context editor.

9 The first contrived error occurs at this point in the sample program. The right parenthesis has been purposely omitted and a carriage return CR has been entered to complete the line.

10 To correct the error in the middle of your program, you must go from the INPUT mode to the EDIT mode. A null line (that is, CR only) entered at the terminal does this switching of modes for you. Simply strike the CR key and a null line is entered. The editor responds with EDIT: and the system is back in the EDIT mode ready to receive editor subcommands.

11 The TYPE subcommand is used to verify the position of the editor's "pointer" at the line that contained the error.

12 To make the correction, enter the CHANGE subcommand with a unique portion of the text that contains the error and the same portion of text repeated but with the error corrected. The portion of text you specify must be unique. If not, later portions of your program that contain the same combination of characters may be inadvertantly changed also. It is assumed that verification is in effect and that the system will retype the corrected line as an additional check. (If you do not have verification in effect, you may issue a VERIFY ON command before you correct this error.)

13 To resume writing your program, enter the INPUT subcommand and the editor reenters the input mode, as the response INPUT: indicates.

14 This statement exceeds 72 characters, the maximum length permitted for FORTRAN source statements. To enter it within the confines of a 72-character line, you must continue the additional portion on the next line preceded by a continuation character (in this case an X) in column 6. You cannot use the TAB key in this situation; you must use the SPACE bar to position the carrier in column 6. In handling literal data of this type, avoid writing, in one statement, a string of characters that will exceed the maximum length of the terminal's printed line, 132.

15 This repetition of the same FORTRAN statement is the second error in the program. It will be corrected after the program is complete but before it is compiled. Continue on.

16 This statement contains an erroneous statement label. It is the last error that has been included in the source program. It has been left uncorrected for the time being so that it will cause the compiler to generate an error message when you attempt to compile this program.

17 With the END statement, you have finished writing your program. Before compiling it, though, check through it for mistakes. Assume that you find only the error mentioned in item 15 above. To correct this error, you must enter the EDIT mode. As you have done before, hit the CR key to enter the edit mode.

18 Issue a TOP subcommand to position the editor's pointer at the beginning of the file. You can now use the FIND subcommand to position the pointer at a line near the one in error. In this situation, the FIND subcommand saves you time since you do not have to

identify each WRITE statement that preceded the one you want, as would be required with the LOCATE subcommand (described later in this section). The line found by the subcommand is typed out for you.

19 The UP subcommand again can be used to move the pointer up to the line you want.

20 The DELETE subcommand removes the extra WRITE statement from your program.

21 Before you can compile this program you must file it (that is, store it) on one of your disks. The FILE subcommand will do this for you. Once your source program is filed on one of your disks under the name MAGICSQ, the ready message signals that you have returned to CMS and you may enter a compiler command.

22 You are now ready to compile your program. A FORTRAN compiler command (in this case FORTGI) specifying the file name MAGICSQ is all that is needed.

23 As you can see, the error that was left uncorrected caused a compiler error message to be typed out and a CMS return code (00008) to be included in the ready message. The return code, in this case, indicates that errors were detected during compilation that may prevent the program from executing. The return code corresponds to the severity code of the compiler error message. message to appear as predicted. The normal compiler error message is typed out, and, in addition, a CMS return code accompanies it.

24 Of course, you must correct the error before the program can be recompiled. Since the file is already on a disk, the EDIT command must be used to regain access to its contents. In response to the command, the CMS enters the EDIT mode and responds with EDIT:.

25 The LOCATE subcommand causes the editor to search your text until a match is found for the character string that you specify. Here, you are searching for the IF statement that contains the undefined label. As you can see, in this case, the first match is not the line you want. Enter the LOCATE subcommand again. The second match is also not the line you want. Enter the subcommand again. The third match locates the right statement. Both the CHANGE and FILE subcommands are used as described previously: CHANGE corrects the error, and FILE replaces the old copy of the program on your disk with the new, corrected copy.

26 Recompile the program using the FORTGI command a second time. This time the compilation is successful and a ready message appears without a return code indicating that no errors were detected in your program.

27 If you would like to test the program and actually produce a magic square, enter the LOAD command and specify the name of the program, MAGICSQ. The program will begin executing. From then on, follow the instructions that are typed out by the magic square program.

28 Since the compiler generates a default name of MAIN for the executable code produced for your program, enter a START MAIN command.

29 When the program has finished executing, a ready message will appear. At this point you may continue using CMS and try some programming on your own or you may log off the system by entering the LOGOUT command. The system responds again with a summary of the time your session used and breaks the connection between your terminal and the computer. The DATA light on the telephone (data set) goes out and your terminal session is over.

Pages 25, 27, 27.2, 27.4, and 28 are foldout pages and have been assembled inside the back cover. These pages should be removed and placed following this page in your manual.

1 *Dial into vm/370*

2 VM/370 ONLINE XXXXXXXXXX

3
```
login eustace
ENTER PASSWORD:

CP WILL BE UP 24 HOURS A DAY
LOGON AT 11.02.09 EST ON THURSDAY 11/30/72
```

4
```
ipl cms
CMS...VERSION 1.0 11/30/72
```

5
```
listfile
FILENAME      FILETYPE      FM
INDIAN        FORTRAN       A1
DUMPREST      ASSEMBLE      A1
SUPERSCR      ASSEMBLE      A1
MY            FORTRAN       A1
INDIAN        TEXT          A1
FORTCLG       EXEC          A1
DUMPREST      LISTING       A1
INDIAN        LISTING       A1
```

6
```
R;
```

7
```
edit magicsq fortran
NEW FILE.
EDIT:
input
INPUT:
```

8
```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c                                                                 c
c                              magicsq                            c
c                                                                 c
c     this is a program for generating a 3-by-3 magic square      c
c                                                                 c
c this section of the program requests the name of the user that wants to generate the  c
c magic square.                                                   c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      write(6,5)
      format(' please enter your name preceded by a blank')
      read(5,10 (CR)
```

9 10 (CR)

11
```
EDIT:
type
      READ(5,10
```

12
```
change /10/10)/
      READ(5,10)
```

13
```
input
INPUT:
10    format('name                    ')
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c this section of the program requests the number for which the user wants the magic    c
c square generated.                                               c
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
15    write(6,20)
      write(6,22)
      write(6,24)
```

14
```
20    format(' enter an integer number of up to 8 digits that is greate
     xr than 14 and divisible by 3')
```

```
22   format(' you must precede it with enough blanks to make up 8 digi
    xts')
24   format(' for example - if your number is 3 digits long precede it
    x with 5 blanks')
25   read(5,30) number
30   format(i8)
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c this section of the program tests the number selected by the user to see if it is      c
c larger than 14 - if not, a message is typed out and the user is asked to enter a new   c
c number.                                                                                c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
     if(number-15) 35, 45, 45
35   write(6,40)
40   format(' sorry, your number is too small')
     go to 15
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c this section of the program tests the number to see if it is divisible by 3 - if not, a  c
c message is typed out and the user is asked to enter a new number.                      c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
45   if(mod(number,3)) 50, 60, 50
50   write(6,55)
55   format(' sorry, your number is not divisible by 3')
     go to 15
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c if the number the user has selected survives the two tests this section of the          c
c program calculates the magic square.                                                   c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
60   ib = number/3-4
     ia = ib + 7
     ic = ib + 5
     id = ib + 2
     ie = ib + 4
     if = ib + 6
     ig = ib + 3
     ih = ib + 8
     ii = ib + 1
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c this section of the program prints out the magic square.                               c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
     write(6,10)
15   write(6,10)
     write(6,65) number
65   format(' here is the magic square for the number ',i8)
     write(6,70) ia, ib, ic
     write(6,70) id, ie, if
     write(6,70) ig, ih, ii
70   format(3(il0))
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c this section of the program asks the user if he wants to enter a new number - if he    c
c answers yes, he is asked to enter a new number - if he answer no, the program ends.    c
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
     write(6,75)
75   format(' want to try again? type in yes or no')
     read(5,80) noquit
80   format(a4)
     data nostop/'yes '/
16   if(noquit.eq.nostop) go to 14
     write(6,85)
```

```
     85   format ( ' thank you for playing - good day' )
          stop
          end
17   (CR)
EDIT:
TOP
TOF:
18   find 65
     65   FORMAT ( ' HERE IS THE MAGIC SQUARE FOR THE NUMBER ',I8 )
19   up 2
          WRITE ( 6,10 )
20   delete
21   file
     R;

22   fortgi magicsq
      G1 COMPILER ENTERED
23    IGI022I         UNDEFINED LABEL
          14
     SOURCE ANALYZED
     PROGRAM NAME = MAIN
     *001 DIAGNOSTICS GENERATED, HIGHEST SEVERITY LEVEL IS 8
     R( 00008 );

24   edit magicsq
     EDIT:
25   locate /if/
      IF ( NUMBER-15 ) 35, 45, 45
     locate /if/
      IF = IB + 6
     locate /if/
       IF ( NOQUIT.EQ.NOSTOP ) GO TO 14
     change /go to 14/go to 15/
      IF ( NOQUIT.EQ.NOSTOP ) GO TO 15
     file
     R;

26   fortgi magicsq
     G1 COMPILER ENTERED
     SOURCE ANALYZED
     PROGRAM NAME = MAIN
     * NO DIAGNOSTICS GENERATED
     R;

27   load magicsq
     R;

28   start main
```

```
EXECUTION BEGINS...
PLEASE ENTER YOUR NAME PRECEDED BY A BLANK
EUSTACE MCGARGLE
ENTER AN INTEGER NUMBER OF UP TO 8 DIGITS THAT IS GREATER THAN 14 AND DIVISIBLE BY 3
YOU MUST PRECEDE IT WITH ENOUGH BLANKS TO MAKE UP 8 DIGITS
FOR EXAMPL - IF YOUR NUMBER IS 3 DIGITS LONG PRECEDE IT WITH 5 BLANKS
     45
 EUSTACE MCGARGLE
 HERE IS THE MAGIC SQUARE FOR THE NUMBER 45

     18   11   16
     13   15   17
     14   19   12
 WANT TO TRY AGAIN? TYPE IN YES OR NO
 no
 THANK YOU FOR PLAYING - GOOD DAY
 R;
```

29

```
 logout
 CONNECT= 00:55:00 VIRTCPU= 000:11.91 TOTCPU= 000:31.40
 LOGOUT AT 11.23.43 EST ON THURSDAY 11/30/72
```

VM/370 Commands for the FORTRAN IV Programmer

The commands listed in Figure 2 represent only a part of the entire VM/370 command language, which is described in the publications IBM VM/370 Command Language User's Guide, Order No. GC20-1804 and *IBM VM/370 EDIT Guide,* Order No. GC20-1805.

These commands were selected because they best meet the typical needs of the FORTRAN IV programmer. System commands that are not normally required or that would be used only by system programmers, system operators, or system maintenance personnel have been omitted. The commands are presented alphabetically, and the underlined portion of the command word identifies the shortest valid abbreviation for that command. Should you require detailed information about the commands listed here, or should the needs of your work require commands that are beyond the scope of this section, consult the publications mentioned above.

CP Commands	CMS Commands and Subcommands	Function
	ACCESS	Activates a virtual disk for the user.
BEGIN		Returns the system to the CMS environment and resumes execution of a program.
	CONVERT[1]	Invokes the SIFT utility.
	CP	Transmits CP commands to the control program without leaving the CMS environment.
	EDIT	Enters the EDIT mode and makes the following subcommands available to the user for file creation and alteration. Entering a null line puts the editor into the INPUT mode.
	BOTTOM	Moves the editor's pointer to the last line of a file.
	CHANGE	Replaces a string of characters with another in one or more lines.
	DELETE	Deletes one or more lines from a file.
	DOWN	Moves the editor's pointer to a subsequent line.
	FILE	Places a file on the user's disk and leaves the EDIT mode.
	FIND	Performs a string search, which is column-dependent, for the specified group of characters. The search begins with the next line or the top of the file if the pointer is at the end of the file.
	FMODE	Changes the filemode of a file.
	FNAME	Changes the filename of a file.
	GETFILE	Includes a part of an existing file in the file being created.
	INPUT	Enters the input mode and accepts subsequent lines as part of the file being created.
	LOCATE	Performs a string search, which is not column-dependent, for the specified group of characters. The search begins with the current position of the editor's pointer.
	NEXT	Moves the editor's pointer to the next line of a file.
	QUIT	Terminates the operation of the editor without effecting any modifications.

Figure 2. VM/370 Commands Frequently Used by FORTRAN Programmers (Part 1 of 3)

	REPLACE	Replaces a line with one or more lines.
	TOP	Moves the editor's pointer to the null line at the beginning of a file.
	TYPE	Types all or part of a file being created.
	UP	Moves the editor's pointer to a previous line of a file.
	VERIFY	Controls the typing of any lines that have been changed or replaced.
	ERASE	Deletes a file from the user's read/write disk.
	EXEC	Executes a file containing one or more CMS commands.
	FILEDEF	Specifies input and output devices and file characteristics to be used by a program during execution.
	FORTGI[1]	Invokes the FORTRAN IV (G1) Compiler.
	FORTHX[1]	Invokes the FORTRAN IV (H Extended) Compiler.
	GLOBAL	Specifies text libraries to be searched in resolving external references in a program that is being loaded.
	GOFORT[1]	Invokes the Code and Go FORTRAN IV Compiler.
IPL		Simulates an initial program load for the user's virtual machine.
	HT	Suppresses the typing of output at the user's terminal.
	HX	Stops the execution of any CMS command and returns the user to the CMS environment.
	INCLUDE	Permits the inclusion of additional TEXT files for use during the execution of a program.
	LISTFILE	Provides a list of all the files that exist on disks that the user has access to during a terminal session.
	LOAD	Loads a TEXT file into storage and establishes the proper linkages for it to be executed.

Figure 2. VM/370 Commands Frequently Used by FORTRAN Programmers (Part 2 of 3)

LOGIN		Identifies the user to VM/370.
LOGOUT		Terminates the terminal session.
	PRINT	Prints a file on the off-line printer in the user's computing center.
	PUNCH	Punches a card deck for a file on an offline card punch unit in the user's computing center.
QUERY	QUERY	Types out the status of the user's virtual configuration and user-defined parameters.
	READCARD	Transfers a virtual card deck from the spooled reader to a disk file.
	RENAME	Changes the filename, filetype, or filemode of a file.
	RT	Restores typing of output, previously suppressed by HT command.
	RUN	Initiates processing, loading, and execution of a source file.
SET	SET	Sets operational characteristics for the user's virtual machine.
	SORT	Sorts the contents of a file.
	START	Begins execution of a previously loaded file.
	STATE	Verifies the existance of a file.
TERMINAL		Sets operational characteristics for the user's terminal.
	TESTFORT[1]	Invokes FORTRAN Interactive Debug.
	TYPE	Types all or a part of a file at the user's terminal.
[1]These commands are available only as program products		

Figure 2. VM/370 Commands Frequently Used by FORTRAN Programmers (Part 3 of 3)

CMS Programming Considerations

A vital aspect of FORTRAN programming under CMS is the creation and management of CMS files. You must create files to hold your FORTRAN source programs. The compiler, in processing your programs, creates files that contain its listing and the executable code it produces. Some of your programs, during their execution, may process or create files containing data. This section describes files in a FORTRAN context. Additional information on CMS file management can be found in the publication *IBM VM/370 Command Language Guide for General Users,* Order No. GC20-1804.

FORTRAN IV Source Files

Before you can invoke the FORTRAN IV (G1), Code and Go FORTRAN IV, or FORTRAN IV (H Extended) compiler, you must have a FORTRAN source program available in a CMS file on one of your disks. The source program is usually created using the CMS EDIT facilities, and may be written in either fixed-form or free-form language format, depending upon the compiler you will be using to compile it. Fixed-form source is acceptable to all the compilers, while free-form source is acceptable to only the Code and Go compiler. You may create these files at your terminal just prior to compiling them or they may be old files, created some time ago, which you want to recompile. In either case, all FORTRAN source files must have a CMS file identifier and file characteristics that conform to the requirements of the compilers.

Identifying FORTRAN Source Files

Every FORTRAN source file that you intend to compile, requires a file identifier with the following format:

filename {FORTRAN / FREEFORT} [*filemode*]

where:

filename - is any valid CMS filename. A valid name consists of from one to eight alphameric characters, which may be any combination of the following:

- Upper Case Letters A through Z
- Lower Case Letters a through z

- Numbers 0 through 9
- National Characters $, #, or @

FORTRAN - is the filetype for a CMS file that contains fixed-length, fixed-form FORTRAN IV records. A FORTRAN record consists of a card image or a line entered at the terminal. Fixed-form records observe the standard column alignment of the FORTRAN IV language (see the publication *IBM System/360 and System/370 FORTRAN IV Language*, Order No. GC28-6515, for a complete description of the FORTRAN language). For example:

```
Column 1      7
       c sample text
         10  d = 10.5
             go to 56
        150  a = b + c*(d + e**f +
            xg + h - 2.*(q + p))
             c = 3
```

Note: Files with a filetype of FORTRAN can also contain fixed-length free-form records, similar in form to these described below under the heading "FREEFORT." However, the use of the FORTRAN filetype for free-form records is not recommended, since it requires inefficient use of disk space and these source files are not acceptable to FORTRAN Interactive Debug. Information on converting these files to variable length files with a filetype of FREEFORT can be found in "Appendix C: SIFT Utility."

FREEFORT - is the filetype for a CMS file that contains variable-length free-form FORTRAN IV records. Here, too, a FORTRAN record consists of a card image or a line entered at the terminal. Free-form records need not observe any column alignment (see the section "Free-Form Input" for a detailed description of preparing free-form FORTRAN statements). For example, the portion of a FORTRAN program shown above in fixed-form would appear like this in free-form:

```
" " sample text
10 d = 10.5
go to 56
150 a = b + c*(d + e**f +-
g + h - 2.*(q + p))
c = 3
```

filemode - is any valid CMS filemode. A valid filemode consists of two characters. The first is alphabetic and corresponds to the name of the disk on which the files resides or is to be placed. The second is numeric and indicates the way in which the disk is to be accessed, that is, read/write, read only, read and erase, or OS simulation. See the publication *IBM VM/370 Command Language Guide for General Users*, Order No. GC20-1804 for detailed information on filemodes.

Characteristics of FORTRAN Source Files

Fixed-Form Files (Filetype of FORTRAN)

Fixed-form FORTRAN files contain records that are 80 characters long. You may type FORTRAN statements or continuation lines of up to 72 characters, including statement numbers. The remaining 8 characters are filled in by CMS with a sequence number.

Free-Form Files (Filetype of FREEFORT)

Free-form FORTRAN files contain variable length records that are a maximum of 81 characters long. The first 8 characters are line numbers supplied by the CMS editor. You may type your FORTRAN statements or continuation lines, including statement numbers, in the remaining 73 characters.

Creating New FORTRAN Source Files

You can easily create new FORTRAN source files at your terminal in either fixed- or free-form using the facilities of the CMS editor.

Creating Fixed-Form Files (Filetype of FORTRAN)

To create a fixed-form source file, type in the EDIT command; specify a unique filename, and assign it a filetype of FORTRAN.

Example:

```
edit newprog fortran
NEW FILE:
EDIT:
input
INPUT:
```

The CMS editor responds to the new filename. The editor recognizes the filetype FORTRAN and will align the subsequent input lines in a fixed FORTRAN format by setting its internal tabs at the correct locations (that is, columns 7 and 10). Striking the TAB key on your keyboard will automatically position the text internally in column 7, regardless of the mechanical tab positions that are set for your terminal. You should, however, set the mechanical tabs on your terminal accordingly, or the terminal listing sheet may not appear as expected or be unreadable. Enter an INPUT subcommand to indicate that you are entering source statements. You may enter your source statements as follows, where the symbols [T], [B], and [R] represent a TAB, SPACE, and RETURN:

```
10[T]          format[B](3f8.2)[R]
[T]            read[B](5,10)[B]p,r,t[R]
[T]            a=p*(1+r/[R]
[B][B][B][B][B]x 100)**t[R]
20[T]          format[B](f8.2)[R]
[T]            write[B](6,20)[B]a[R]
[T]            stop[R]
[T]            end[R]
[R]
EDIT:
file[R]
R;
```

Note: For continuation lines, do not tab to column 7 and backspace to column 6. Use the space bar to space to column 6 from the beginning of the line.

The statements shown above are arranged by the editor into the following fixed FORTRAN format.

```
10      FORMAT (3F8.2)
        READ(5,10)P,R,T
        A = P*(1 + R/
       X 100)**T
20      FORMAT(F8.3)
        WRITE(6,20) A
        STOP
        END
```

When you have completed your program or when you need to make corrections, be sure to hit two carrier returns. The first return ends the line you are entering, and the second indicates that you have reached the end of your input and want to enter editor subcommands. If you do not hit the second return, any subsequent editor subcommands that you may enter will be treated as additional lines in your program. The full range of editor subcommands can be used to modify your source code. When you have completed your program, it must be filed using the FILE subcommand before you can enter the command required by the FORTRAN compiler you are going to use. See the sections in this book that describe how to use your particular compiler.

Creating Free-Form Files (Filetype of FREEFORT)

To create a free-form source file, type in the EDIT command; specify a unique filename, and assign it filetype of FREEFORT.

Example:

```
edit newprog2 freefort
NEW FILE:
EDIT:
input
INPUT:
00000010
```

The CMS editor responds to the new filename and recognizes the filetype of FREEFORT. Enter an INPUT subcommand to indicate that you are entering source statements. The editor prompts you with a line number. This line number can be used to locate a line when editing your file and is acceptable to FORTRAN Interactive Debug, a program product that is available for debugging Code and Go programs. Do not confuse this number with the FORTRAN statement number. You must still supply a statement number for any statements that require them (for example, FORMAT statements, target statements of a GO TO or IF statement, and the end of range statements in DO loops). You may enter your FORTRAN statements immediately after the CMS-supplied line number, as follows without regard for column alignment or tab settings:

```
00000010    " "sample program
00000020    10 format (3f8.2)
00000030    read (5,10) p,r,t
00000040    a = p*(1 + r/-
00000050    100)**t
00000060    20 format (f8.2)
00000070    write (6,20) a
00000080    stop
00000090    end
00000100

EDIT:
file
R;
```

Note: The space between the line numbers and the FORTRAN statements has been inserted for clarity; it is not required and can be omitted. Since the CMS escape character is a quote, a second quote is required at the beginning of the comment line. An alternative would be to change the escape character.

When you have completed your program or when you need to make a correction, be sure to hit two carrier returns. The first return ends the line you are entering, and a new line number will be typed out. The second return, immediately after the line number, indicates that you have reached the end of your input and want to enter editor subcommands. If you do not hit the second return, any subsequent editor subcommands that you may enter will be treated as additional lines in your program. The full range of editor subcommands can be used to modify your source code. When you have completed your program, it must be filed using the FILE subcommand before you can enter the GOFORT compiler command for the Code and Go compiler. See the section of this book that describes how to use the Code and Go compiler.

Preparing to Compile FORTRAN Source Programs

If you have an existing FORTRAN source program that is already filed in your system, and you wish to compile it, first, make sure that it has a filetype of FORTRAN or FREEFORT. (Remember, that only Code and Go accepts both FORTRAN and FREEFORT filetypes.) Use the LISTFILE command specifying the name of the file you want. This will type a list of the file

identifiers that you have already used. If necessary, change the filetype, as in the example below, with the RENAME command and then issue the appropriate command (in this case FORTGI) to begin compilation.

Example:

```
listfile  oldprog  *
FILENAME      FILETYPE      MODE
OLDPROG       SOURCE        A1
R;

rename oldprog source * oldprog fortran *
R;

fortgi oldprog
G1 COMPILER ENTERED
SOURCE ANALYZED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
R;
```

If you have just created your FORTRAN program and assigned it the filetype FORTRAN or FREEFORT in your EDIT command, you need only issue the appropriate compiler command.

Example:

If all the defaults are acceptable:

```
fortgi newprog
```

or, if any of the defaults are to be changed:

```
fortgi newprog (bcd print id list)
```

CMS Return Codes Following Compiler Commands

CMS produces a return code following the execution of a FORTRAN compiler command. It appears in the ready message and corresponds to the highest diagnostic message severity level encountered during the compilation.

Example:

```
R(00004);
```

The meaning of the return codes are shown in Figure 3.

Code	Meaning
00000	No errors were detected. There may be warning messages, however. If a warning message is produced, check your program for possible errors, as the reliability of your results may be in doubt.
00004	Possible errors were detected or warning messages were issued. Execution of your program should be successful, but the result may not be reliable.
00008	Errors were detected. Compilation continues but execution may fail. If specified, an object module will be created but you may not be able to execute it.
00012	Severe errors were detected. Compilation may not continue; if it does, execution of the program is impossible.
00016	Extremely severe errors were detected. Compilation terminates at the point at which the error was detected.

Figure 3. CMS Return Codes for FORTRAN Compilations

Entering FORTRAN Source Files from Devices other than the Terminal

You may have FORTRAN source files on tape or punched cards. To make these files known to the system, you must issue a FILEDEF command whose ddname is FORTRAN and which specifies the appropriate device type.

Examples:

To use a source file on tape, issue the following FILEDEF:

```
filedef fortran tapn
```

where:

n is a number from 1 through 4 that corresponds to virtual tape units 181 through 184.

To use a source file on a deck of cards, issue the following FILEDEF:

```
filedef fortran reader
```

FORTRAN Compiler Output Files

As a result of a compilation, two files may be produced with filetypes of LISTING and TEXT. These files are placed on the same disk as your FORTRAN source file. If this source disk is read only (R/O), the system determines if the disk containing your source program is a read only (R/O) extension of a read/write (R/W) disk. If it is an extension, the system will attempt to put the LISTING and TEXT files on the R/W parent disk. If this alternative fails, the system will try to put the output files on your primary

(A) disk. Should that also fail, an error condition exists, and a message is typed at your terminal. You must either issue an ACCESS command to make available a disk on which the system can place the compiler output files or specify compiler options that will not produce these files before you can reissue a compiler command.

LISTING File

The LISTING file is a CMS disk file that can be optionally produced by your compiler. The file contents depend upon the compiler command options specified. Figure 4 below indicates how the LISTING file can be created for your particular compiler.

Compiler	When a LISTING File is Produced
FORTRAN IV (G1)	Always produced, unless the NOPRINT option is specified with the FORTGI command.
Code and Go FORTRAN IV	Produced for error messages or when the SOURCE option is specified with the GOFORT command.
FORTRAN IV (H Extended)	Always produced unless the NOPRINT option is specified with the FORTHX command. Produced on the offline printer, instead of the primary disk, when the PRINT compilation option is specified.

Figure 4. Producing a LISTING File with Various Compilers

When produced, CMS gives the LISTING file the same filename as your source program but a filetype of LISTING. This file collects all the information that is usually included in a compiler output listing.

Obtaining a Printed Copy of Your LISTING File

Normally, the LISTING file is written on a disk (since DISK is the default option for the compiler commands). Should you wish a printed copy of this file, you need only issue a PRINT command to obtain a listing on the off-line printer in your computing center. You can, in addition, issue a TYPE command to examine the contents of the file at your terminal.

Example:

```
print newprog listing
```

or:

```
type newprog listing
```

If you do not want to place a copy of your LISTING file on disk and only want a printed copy, you must specify the PRINT option with your compiler command.

Example:

```
fortgi newprog ( list print )
```

Retaining LISTING Files

All LISTING files are placed on one of your accessible read/write disks unless the PRINT or NOPRINT options are in effect. These files will remain there until you delete them or they are replaced by the new LISTING file when you recompile the same source program. (When the options specified for the new compilation do not produce a LISTING file on disk or when the program abnormally terminates, the old LISTING file may not be replaced.) If you want to keep a permanent copy of old LISTING files on your disks, it is advisable to rename them with any unique filetype before the next compilation is begun. You can use the RENAME command for this.

Example:

```
fortgi newprog ( list )
G1 COMPILER ENTERED
SOURCE ANALYZED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
R;

listfile newprog *
FILENAME              FILETYPE              FM
NEWPROG               FORTRAN               A1
NEWPROG               TEXT                  A1
NEWPROG               LISTING               A1
R;

rename newprog listing * newprog oldlist *
R;

fortgi newprog ( list )
G1 COMPILER ENTERED
SOURCE ANALYZED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
R;

listfile newprog *
FILENAME              FILETYPE              FM
NEWPROG               FORTRAN               A1
NEWPROG               OLDLIST               A1
NEWPROG               TEXT                  A1
NEWPROG               LISTING               A1
```

TEXT File

The TEXT file is a CMS disk file that can be optionally produced by your compiler depending upon the options specified with the compiler command. Figure 5 below indicates when the TEXT file is created for your particluar compiler.

Compiler	When a TEXT File is Produced
FORTRAN IV (G1)	Always produced unless the NOLOAD option is specified with the FORTGI command
Code and Go FORTRAN IV	Whenever the DECK or TEST option is specified with the GOFORT command
FORTRAN IV (H Extended)	Always produced unless the NOOBJECT option is specified the the FORTHX command

Figure 5. Producing a TEXT File with Various Compilers

When produced, CMS gives the TEXT file the same filename as your source program but a filetype of TEXT. The file contains the executable code that is created from your FORTRAN source program. The TEXT files produced under CMS are identical to object programs produced under OS. The code contained in the TEXT file may be loaded and executed under CMS or transferred, link edited, and executed under OS.

For more information on loading and executing TEXT files under CMS, refer to the appropriate section in this book describing your compiler; for information on link editing and executing object programs under OS, see the OS programmer's guide appropriate for your compiler.

Identifying Programs in a TEXT File

The entry point name for a main program in a TEXT file is the name you specified for the NAME option of the compiler command or its default. Subprograms have the entry point name that you specified in the FORTRAN SUBROUTINE statement. A main program that follows a subprogram has the name MAIN.

The copy of the TEXT file pseudo-assembler listing that is included in your LISTING file contains an identification for the programs in it. Columns 73-76 of each line of code contain four characters that identify whether that code was generated for a main program or subprogram as follows:

- Main Programs -- The first four characters of the name specified by the compiler NAME option or the letters MAIN.

- Subprograms -- The first four characters of the name specified in the SUBROUTINE statement.

Retaining TEXT Files

All TEXT files are placed on one of your accessible read/write disks. The criteria to determine which disk will be used is the same as that for the LISTING file. These files will remain there until you delete them or they are replaced by the new TEXT file that is produced when you recompile the same source program. (When the options specified for the new compilation do not produce a TEXT file on disk or when the program abnormally terminates, the old TEXT file may not be replaced.) Therefore, if you want to keep a copy of old TEXT files on your disks, it is advisable to rename them with any unique filetype before the next compilation is begun. You can use the RENAME command for this.

Example:

```
fortgi newprog ( load )
G1 COMPILER ENTERED
SOURCE ANALYSED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
R;

listfile newprog *
FILENAME              FILETYPE              FM
NEWPROG               FORTRAN               A1
NEWPROG               LISTING               A1
NEWPROG               TEXT                  A1
R;

rename newprog text * newprog oldtext =
R;

fortgi newprog ( load )
G1 COMPILER ENTERED
SOURCE ANALYSED
PROGRAM NAME = MAIN
* NO DIAGNOSTICS GENERATED
R;

listfile newprog *
FILENAME              FILETYPE              FM
NEWPROG               FORTRAN               A1
NEWPROG               OLDTEXT               A1
NEWPROG               TEXT                  A1
NEWPROG               LISTING               A1
R;
```

A TEXT file can be produced by the FORTRAN IV (G1), Code and Go FORTRAN IV and FORTRAN IV (H Extended) compilers. The file contains the executable codes in 80 column card format. There are four types of 80 column card formats. These are identified by the characters ESD, RLD, TXT, or END in columns 2 through 4. Column 1 of each card format contains a 12-2-9 punch. Columns 73 through 80 contain the first four characters of the program name followed by a four-digit sequence number. The remainder of the card contains program identification.

ESD CARD: ESD cards describe the entries of the External Symbol Dictionary, which contains one entry for each external symbol defined or referred to within a module. For example, if program MAIN calls subprogram SUBA, the symbol SUBA will appear as an entry in the *Symbol Dictionaries* of both the program MAIN and the subprogram SUBA. CMS matches the entries in the dictionaries of other included subprograms, and when necessary, to the library.

ESD cards are divided into four types, and are identified by the digits 0, 1, 2, or 5 in column 25 of the first entry in the card, column 41 if a second entry, and column 57 is a third entry (there can be 1, 2, or 3 external symbol entries in a card). The ESD types are described in Figure 6.

ESD	Contents
0	Name of the program or subprogram, and indicates the beginning of the module. It will assume the FORTRAN default value of MAIN if you have not specified a name in the compiler command. The name of a subprogram will come from the SUBROUTINE, FUNCTION, or BLOCK DATA statement.
1	Entry point name appearing in an ENTRY statement of a subprogram
2	Name of a subprogram referred to by the source module through CALL statements, EXTERNAL statements, and explicit and implicit function references. (Some usages of FORTRAN are of such complexity, that they call in a function subprogram instead of generating in-line coding; these are *implicit function references)*
5	Information about a COMMON block.

Figure 6. Types of ESD Card Formats in FORTRAN TEXT Files

TXT CARD: TXT cards contain the constants and variables used by the programmer in his source module, any constants and variables generated by the compiler, coded information for FORMAT statements, and the machine instructions generated by the compiler from the source module.

RLD CARD: RLD cards describe entries in the *Relocation Dictionary* which contain one entry for each address that must be resolved before a module can be executed. The Relocation Dictionary contains information that enables absolute storage addresses to be established when a module is loaded into main storage for execution. RLD cards contain the storage address of subprograms called by ESD type 2 cards.

END CARD: The END card indicates the end of the object module, the relative location of the main entry point, and the length (in bytes) of the object module.

Figures 7 and 8 show typical deck structures for the FORTRAN compilers under CMS.

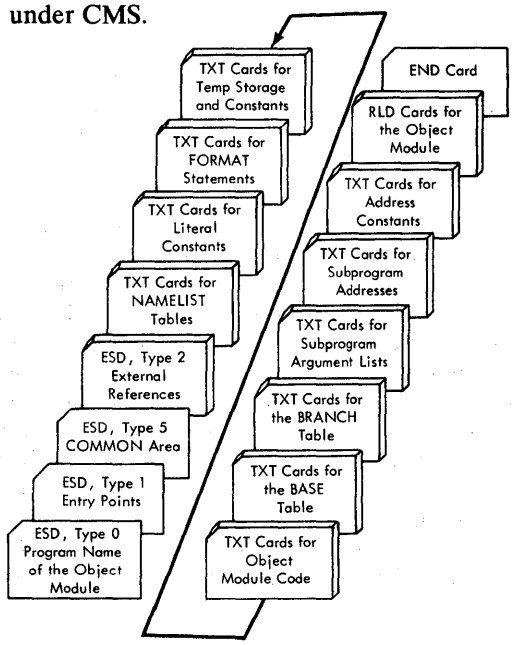


Figure 7. TEXT File Structure Produced by the FORTRAN IV (G1) and Code and Go Compilers

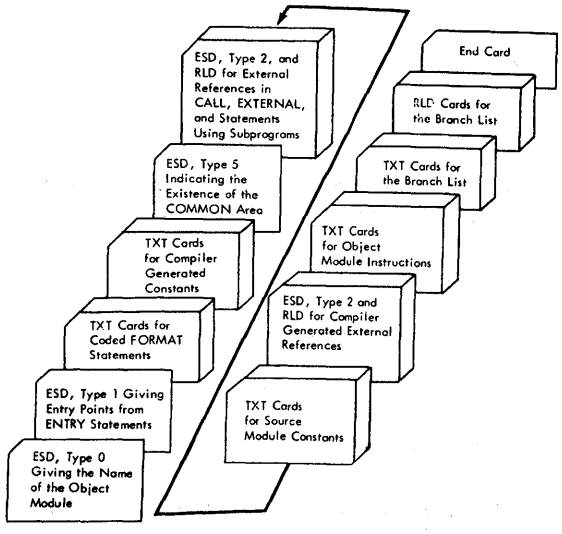


Figure 8. Object Module Deck Structure Produced by the FORTRAN IV (H Extended) Compiler

Execution-Time Input and Output Files

In using TEXT files to do your work, it may be necessary to provide them with information or have your programs create data. Information can be presented to and retrieved from your programs through these execution-time files.

There are two methods by which execution-time files can be organized and processed: sequential and direct access. Sequential files, as the name implies, store records in sequential order. The first record that is written into the file occupies the first position in the file. Each succeeding record follows in order. The need for sequential files is usually dictated by the type of input and output devices that you are using. Devices such as terminals, tape units, printers, and card punches can, by design, only handle sequential records. Disk units can also accept records in a sequential order; however, their design is more suited for direct access records. Moreover, direct-access units are the only devices that can accept direct access files. In direct access files, records are not positioned in a sequential order. The first record written in the file may not be in the first position, since records are placed in any available space, and pointers are kept to indicate where the parts of the file reside.

Whether a file is sequential or direct access will, to a great extent, determine how a record is defined, the way it is identified, and how it is referred to in a FORTRAN program. The following discussion describes how files are defined through the use of the CMS FILEDEF command, and how they are identified to the system through the use of file identifiers and FORTRAN input and output statements.

Defining Execution-time Files

All execution-time files that you want to use must be defined to CMS with a FILEDEF command. Three files are pre-defined for you. They are:

Sequential

- Terminal Input
- Terminal Output
- Punched Card Output

An initialization routine supplies the FILEDEF for these files. You must define all the other files that you want to use. They include:

Sequential

- Disk Input
- Disk Output

- Tape Input
- Tape Output
- Punched Card Input
- Printed Output

Direct Access

- Disk Input
- Disk Output

You must supply your own FILEDEF command for any of these files used by your program. In addition, you may change or replace the three pre-defined files by supplying FILEDEF commands of your own for them. Regardless of the type of file you are using, there are several general guidelines that must be followed in defining and using files.

- Each file that you use in your program must be defined to the system (either through a system-supplied definition or one that you supply).
- Do not use the same definition for more than one file.
- Do not use the same file on more than one type of device in the same program.
- You may refer to the same file from more than one program through different devices and access methods, if you change the file and its definition appropriately before using it.

Pre-defined Files

Three pre-defined files are provided for you by the FORTRAN initialization routine. These files are recognized by CMS when you refer to them in FORTRAN input or output statements with the FORTRAN data set reference number that has been assigned to them. Since they are pre-defined you do not have to supply a FILEDEF command for them, unless you want to change their definition or create new files to replace them. (See the section "User-defined Files" for more information.) Figure 9 summarizes the data set reference numbers that are assigned to pre-defined files, the FORTRAN input and output statements in which they can be used, the devices utilized, and the maximum lengths of the records in the file.

FORTRAN Data Set Reference Numbers	Used in the Following FORTRAN Input and Output Statements	Identifies	Requires Records of the Following Format and Maximum Length
5	READ (5,b) list READ (5,*) list	Terminal Input	Fixed-length, unblocked (F), 130 characters long
6	WRITE (6,b) list WRITE (6,*) list	Terminal Output	Fixed-length, unblocked (F), 131 characters long
7	WRITE (7,b) list	Punched Card Output	Fixed-length, unblocked (F), 80 characters long
Notes:	In the input and output statements the variables are: b - FORMAT statement number list - series of variables and array names		

Figure 9. Summary of Data Set Reference Numbers, Input/Output Statements, and Record Formats Used for Pre-Defined Files

Pre-defined Terminal Input Files

You can use the FORTRAN list-directed or sequential READ statements to read data from your terminal. Data set reference number 5 is available as the default for terminal input. If you use list-directed input and output, refer to the section "List-directed Input and Output" for more information. If you use formatted or unformatted input and output, you should construct a mechanism (called self-prompting) in your program to notify yourself that the program is ready to read data from the terminal. Add, to your program, a FORMAT statement with a literal field that reminds you what data to enter. Following the FORMAT statement, include a WRITE statement that will type the literal at your terminal. Both these statements should precede the list-directed or sequential READ statement that will actually read the data you want to enter. When your program executes, it will first type out your reminder at the terminal; it will then wait until you enter your data before continuing. Remember, if you use formatted input, you must type in your data so that it corresponds to the specifications of the FORMAT statement that controls it.

Example:

The following statements in your program	and the following data entered at your terminal	produce the following results at the terminal

`   WRITE (6,10)`
`10 FORMAT (' A=?')`
`   READ (5,20) A`
`20 FORMAT (F8.3)`
`   A=A**2`
`   WRITE (6,30) A`
`30 FORMAT (' A=',F8.3)`
	`00003.4`	`A=?`
		
		`A=11.560`

Pre-defined Terminal Output Files

You can use FORTRAN list-directed or sequential WRITE statements to type at your terminal, data created by your FORTRAN program. Data set reference number 6 is available as the default for terminal output.

Example:

These statements in your program	produce the following at your terminal

`20 FORMAT(F8.3)`
`A=14.2`
`WRITE(6,20)A`
	`14.200`

All terminal output operations include ASA carriage control characters. Each output line must be preceded by either a blank or a special character or a FORMAT statement must supply the required character. Figure 10 illustrates the variety of terminal printer positions that are available.

Character	Printed Format	Printer Action
blank	Single spaced lines	The carrier is advanced one line A line is printed
0	Double spaced lines	The carrier is advanced one line The carrier is advanced a second line A line is printed

Figure 10. ASA Carriage Control Characters

Pre-defined Punched Card Output Files

You may have data created by your program punched into a card deck with the FORTRAN sequential WRITE statement. Data set reference number 7 is available as the default for punched output. The data to be punched will be placed on a spooled punch file and subsequently punched into a deck of cards on an off-line card punch device. The system identifies your card deck with a header card that contains your user identification.

User-defined Files

User-defined files may already exist in your system, containing data that you will want your program to process. Conversely, you may want your program to create a file to hold data that was generated during its execution. Since they are not pre-defined, they cannot be identified by CMS and associated with your program. You must define all files, whether new or old, that use the following access methods and devices:

Sequential

- Disk Input
- Disk Output
- Tape Input
- Tape Output
- Punched Card Input

Direct Access

- Disk Input
- Disk Output

You may, in addition, define the following files to be used in place of the system's pre-defined files or change them to suit your own needs:

Sequential

- Terminal Input
- Terminal Output
- Punched Code Output

To make user-defined files accessible to your programs, you must establish links to them through the CMS FILEDEF command used in conjunction with the data set reference numbers in your FORTRAN input and output statements and the identifier of the file that you want to use or create.

FILEDEF Command for FORTRAN Programmers

Figure 11 illustrates the general form of the FILEDEF command of interest to you, the FORTRAN programmer.

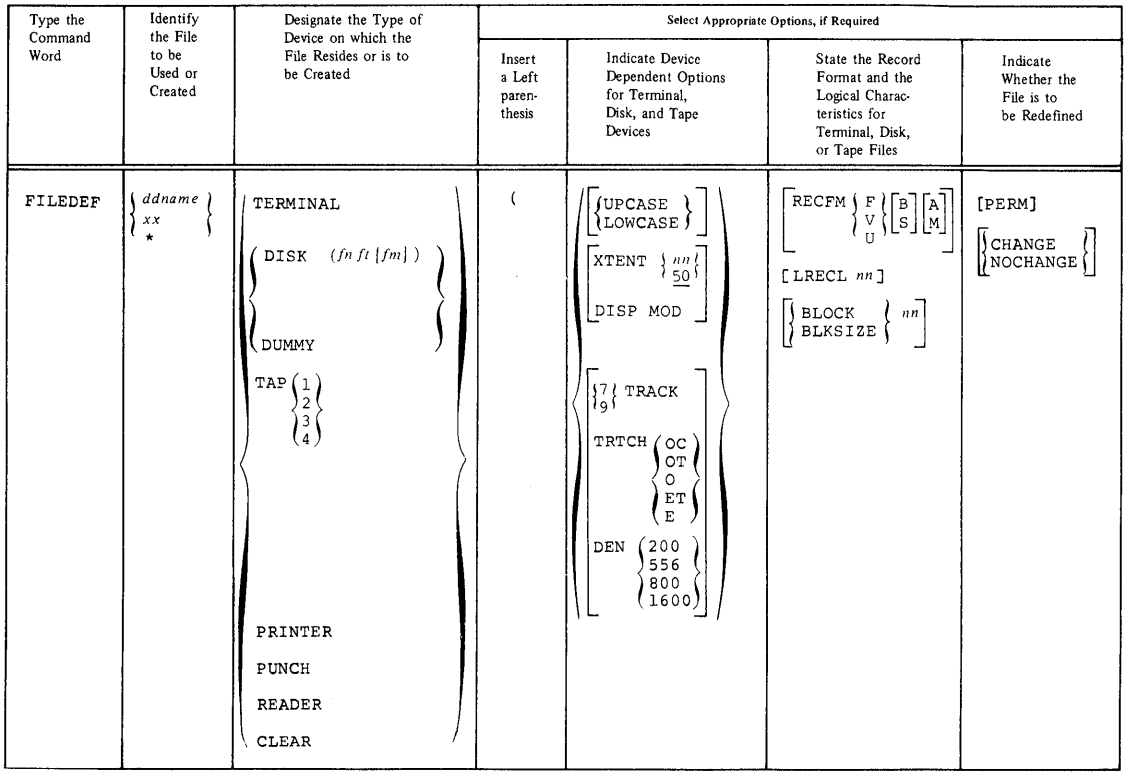


Figure 11. General Form of the FILEDEF Command for FORTRAN Programmers

Command Word

FILEDEF — This is a required part of the command and must always be typed. If you do not include any options after the command word, all current FILEDEF commands in effect will be typed at your terminal.

Options

- Establishing a Link Between Input/output Statements and Files

ddname — The data definition name provides the link between your FORTRAN input and output statements and the file that you want to process or create. The standard FORTRAN data definition name has the following format:

FT*xx* F*yyy*

where:

xx is a FORTRAN data set reference number. You must use this number in any FORTRAN input or output statements that refer to the file being defined. You may not specify 00 as a data set reference number. This number may range from 01 to a maximum value that was determined for your system when it was installed. Consult with your system administrater for the maximum number available to you.

yyy is a sequence number (ranging from 001 to 999) that identifies multiple files under the same data set reference number. For direct access files this number is always 001. For sequential files this number will vary depending upon the order in which the file is referred to in your program. See the section "Using Multiple Sequential Files" for more information.

xx A FORTRAN data set reference number used alone performs the same function as the ddname; however, it cannot be used for multifiles, since it generates a default ddname of FT*xx*F001. The data set reference number specified must be used in your FORTRAN input and output statements referring to the file being defined.

- Describing the I/O Device on Which Your File is to Reside

TERMINAL This option specifies that your user-defined file is to be read or written at the terminal. By using this option, you can either change the characteristics of the system's pre-defined file or create an entirely new file for the terminal.

UPCASE

Specifies that the data entered at the terminal will appear on the printout in upper case characters. This is the default for the option.

LOWCASE

Specifies that the data entered at the terminal will appear on the printout in lower case characters.

PRINTER This option indicates that your user-defined file is to be written on an off-line printer. Files defined as PRINTER may only be used for output.

PUNCH This option indicates that your user-defined file is to be punched into a card deck. Files defined as PUNCH may only be used for output.

READER This option indicates that you want to read a user-defined file consisting of a deck of cards. Files defined as READER may only be used for input.

DISK

This option indicates that you want to use an existing disk file or want to create a new one.

fn ft [fm]

If you are reading an existing file you must include its file identifier. If you are creating a new disk file, you may supply your own file identifier for it; if you do not, the system will supply the following default file identifier:

FILE FT*xx*F*yyy* A1

For files that are defined as spanned (VS or VBS) the file mode must be specified as 4.

XTENT{*nn*|50}

This option must be included in each FILEDEF command that defines a FORTRAN direct access file (that is, a file that requires a FORTRAN DEFINE FILE statement). The variable *nn* should correspond to the number of records that you specified in the DEFINE FILE statement. If you do not include this option, a value of 50 records is assumed.

DISP MOD

This option indicates that the read/write pointer is to be positioned after the last record in the disk file.

DUMMY

This option may be used in place of the DISK option only. It indicates that no real input or output operation is to be performed for a disk file. You may also specify any of the disk options; however, they will be ignored.

TAP*n*

This option indicates that you want to use an existing tape file or create a new one. The variable *n* represents the symbolic tape number and can range from 1 through 4. The numbers correspond to tape units attached to your virtual machine addresses 181 through 184.

*n*TRACK

This option indicates the type of tape device being used. *n* represents the number of tracks that the tape device records on. Specify either 7 or 9 for the variable *n*.

TRTCH *aa*

This option is used for seven track tapes to indicate the recording technique that is being used. The variable *aa* is a code that specifies the parity, converter, and translator settings. Figure 12 lists the possible technique specifications that are available for this option.

aa	parity	converter	translator
OC	odd	on	off
OT	odd	off	on
O	odd	off	off
ET	even	off	on
E	even	off	off

Figure 12. Tape Recording Technique Specification Available for the TRTCH Option of the FILEDEF Command

DEN *nnnn*

This option indicates the density of the tape being used. The variable *nnnn* may specify one of the following densities: 200, 556, 800, or 1600. If the *n*TRACK option has not been included, a density of 200 or 556 assumes a default of 7TRACK and a density of 800 or 1600 assumes a default of 9TRACK.

- Specifying the Format and Logical Characteristics of Your File

RECFM *aaa[a]* This option indicates the format of the records being read or written and whether they contain carriage and print control characters. The variables *aaa* and *a* are codes that represent the possible record formats and control characters. Figures 13 and 14 list the possible record formats and control characters available for this option. The default RECFM is fixed-length records.

aaa	Record Format
F	fixed-length records
FB	fixed-length, blocked records
V	variable-length records
VB	variable-length, blocked records
U	undefined-length records
FS	fixed-length, standard block records
FBS	fixed-length, blocked, standard block records
VS	variable-length, spanned records
VBS	variable-length, blocked, spanned records

Figure 13. Record Formats Available for the RECFM Option of the FILEDEF Command

a	Control Characters
A	ASA carriage control characters
M	machine control characters

Figure 14. Control Character Specifications Available for the RECFM Options of the FILEDEF Command

See the section "Specifying FORTRAN Record Formats and Logical Characteristics under CMS" for information on using the RECFM option.

LRECL *nn* This option indicates the length, in bytes, of the logical records in a user-defined file. The record format, which you specified in RECFM above, determines, how you must specify LRECL. Figure 15 lists the criteria for determining logical record lengths. The default LRECL is 80.

For RECFM	LRECL Must
F, FB, FS, or FBS	Specify the actual size of the records
V, VB, VS, or VBS	Specify the size of the longest record
U	Be omitted
The maximum LRECL that can be specified is 65K bytes.	

Figure 15. Criteria for Determining a Value for the LRECL Option of the FILEDEF Command

See the section "Specifying FORTRAN Record Formats and Logical Characteristics under CMS" for information on using the LRECL option.

{BLOCK / BLKSIZE} *nn* This option indicates whether records are to be read or written individually or in groups. It also establishes the size of the group of records. Here, too, the record format specified in RECFM and the value specified in LRECL determine the value you must specify for BLOCK. Figure 16 lists the criteria for determining block sizes. The default BLOCK is 80.

For RECFM	BLOCK must
F or FS	specify the same value as LRECL
FB or FBS	specify a multiple of LRECL
V or VS	specify the value of LRECL plus four bytes for a segment descriptor word.
VB or VBS	specify the value of LRECL plus four bytes for the seqment descriptor word of each record that can be contained in the block plus four bytes for a block descriptor word.
U	specify the greatest amount of space required to hold all the records that are to be grouped together.
The maximum BLOCK value that can be specified is 65K bytes.	

Figure 16. Criteria for Determining a Value for the BLOCK Option of the FILEDEF Command

See the section "Specifying FORTRAN Record Formats and Logical Characteristics under CMS" for information on using the BLOCK or BLKSIZE option.

PERM — This option indicates that the file characteristics specified in a FILEDEF command are to remain in effect until they are either explicitly cleared or changed with a new FILEDEF command that has the CHANGE option.

CHANGE — This option indicates that if a file definition exists for the ddname that is specified in this command, the options that are included will replace the corresponding options in the old FILEDEF command. This is the default if PERM, CHANGE, or NOCHANGE are not specified.

NOCHANGE — This option indicates that if a file definition exists for the ddname specified in this FILEDEF command, the options that are included will not replace the corresponding options in the old FILEDEF command.

Specifying FORTRAN Record Formats and Logical Characteristics under CMS

The following options of the FILEDEF command describe the format and logical characteristics of FORTRAN records that are read from or written into a CMS file during the execution of your program:

- RECFM - indicates the format of a set of FORTRAN records.
- LRECL - indicates the maximum length of a FORTRAN record.
- BLOCK or BLKSIZE indicates the maximum amount of space required by one or more FORTRAN records that are to be read or written by a single input or output statement.

FORTRAN records whose format and logical characteristics are described by the above FILEDEF options can be transferred into and out of virtual storage under the control of a FORMAT statement (formatted I/O) or without a FORMAT statement (unformatted I/O, including NAMELIST and list-directed input and output).

For formatted I/O, it is advisable under CMS to define your files as fixed-block (FB) record format, with a logical record length of 80, and a block size of 800. This is advantageous as it makes your execution-time files acceptable to the CMS editor. For unformatted I/O, you must define your files as variable-length, spanned (VS) or variable-length, blocked, spanned (VBS). In addition, the filemode for these files must be x4. If you are using unformatted I/O or should you need to define files for use on an OS system, see Appendix E for more information.

Identifying and Using User-defined Files

Sequential Files

User-defined Disk Input and Output Files

Each sequential disk file, whether it is used as input or output for a FORTRAN object program, requires a file identifier with the following format:

filename filetype [filemode]

where:

filename - is any valid CMS filename.

filetype - is any valid CMS filetype. It is recommended that you also use the ddname for the filetype. The ddname is the default filetype for FORTRAN execution-time files and will remind you what data set reference number has been defined for the file.

filemode - is optional, but may specify any user disk (A through G). These disks may be any available mode on input (1 through 5); however, they can only be modes 1, 4, or 5 for output. If no mode is specified, A1 is assumed. Files that contain spanned records (that is, RECFM is VS or VBS) must have a mode of 4.

Sequential disk files are associated with your program through the ddname, device specification of DISK, and file identifier that you specified in the FILEDEF command that defines them. To define your own sequential disk files you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx* F*yyy* DISK *filename filetype [filemode][options]*

where:

xx - is any FORTRAN data set reference number acceptable to your system.

yyy - is any valid sequence number, 001 if you are not using multiple files or any number form 001 to 999 if you are using multiple files. (See the section "Using Multiple Files" for more information.)

filename - the name of the file to be used or created. If you omit the filename, the system assumes FILE.

filetype - the type of the file to be used or created. If you omit the filetype, the system assumes the ddname of FT*xx* F*yyy*.

filemode - is optional, but if specified is the filemode of the file to be used or created.

options - are any valid FILEDEF options for sequential disk files.

Figure 17 illustrates how the FILEDEF command associates sequential disk file with the input and output statements in your FORTRAN programs.

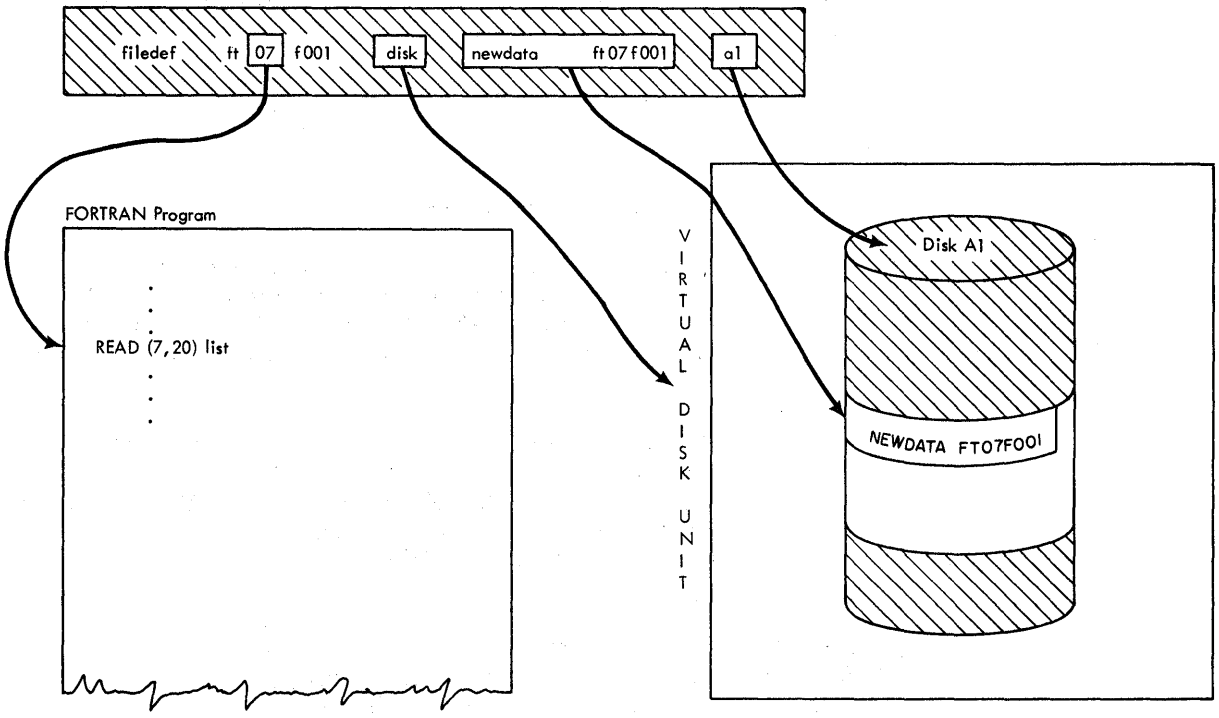


Figure 17. FILEDEF Command for User-defined Sequential Disk Files

For input operations, the system searches for the file identifier specified. For output operations into an existing file, the system places new data at the end of the file. For output operations into a new file, the system places the new data onto your disk and creates for it the file identifier that you specified in the FILEDEF command.

User-defined Tape Input and Output Files

Tape files are associated with your program through the ddname and device specification of TAP *n* in the FILEDEF command that defines them. To define your own tape files, you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT *xx* F *yyy* TAP *n* [*options*]

where:

xx - is any FORTRAN data set reference number acceptable to your system.

yyy - is any valid sequence number, 001 if you are not using multiple files or any number from 001 to 999 if you are using multiple files. (See the section "Using Multiple Files" for more information.)

n - is any valid tape unit (1 through 4).

options - are any valid FILEDEF options for tape files.

Figure 18 illustrates how the FILEDEF command associates tape files with the input and output statements in your FORTRAN program.

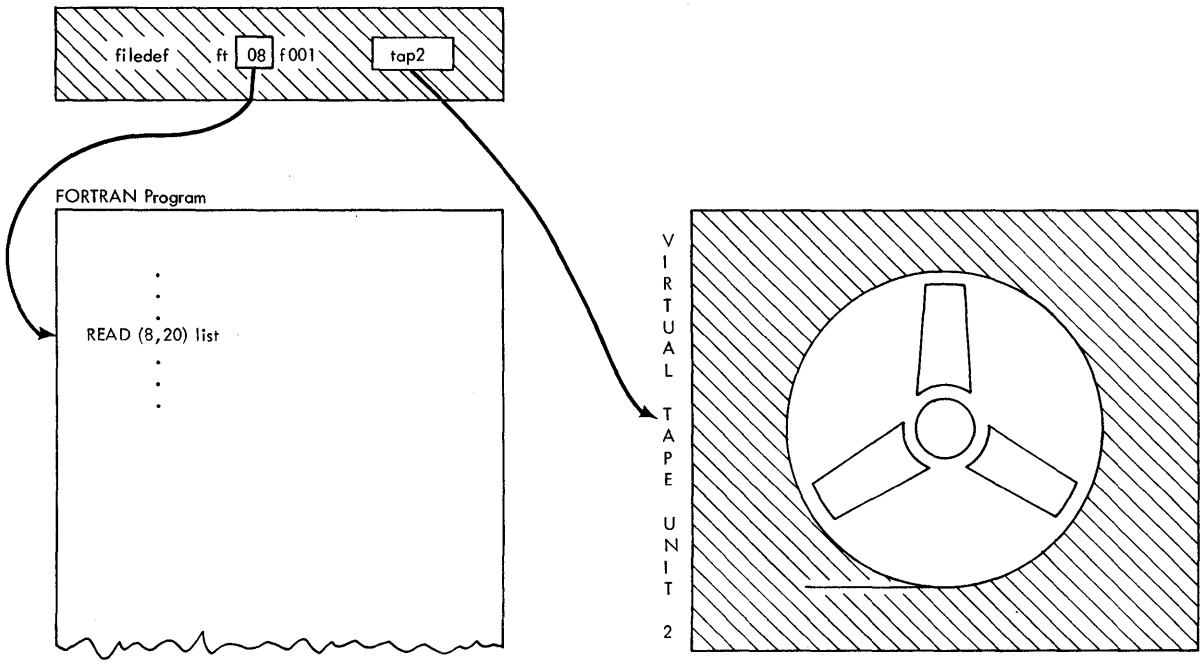


Figure 18. FILEDEF Command for User-defined Tape Files

For input and output operations, the system uses the tape that is mounted on the tape unit specified in the FILEDEF command.

User-defined Terminal Input and Output Files

Terminal files are associated with your program through the ddname and device specification of TERMINAL in the FILEDEF command that defines them. To define your own terminal files, you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx* F001 TERMINAL [*options*]

where:

xx - is any FORTRAN data set reference number acceptable to your system.

options - are any valid FILEDEF options for terminal files.

Figure 19 illustrates how the FILEDEF command associates terminal files with the input and output statements in your program.

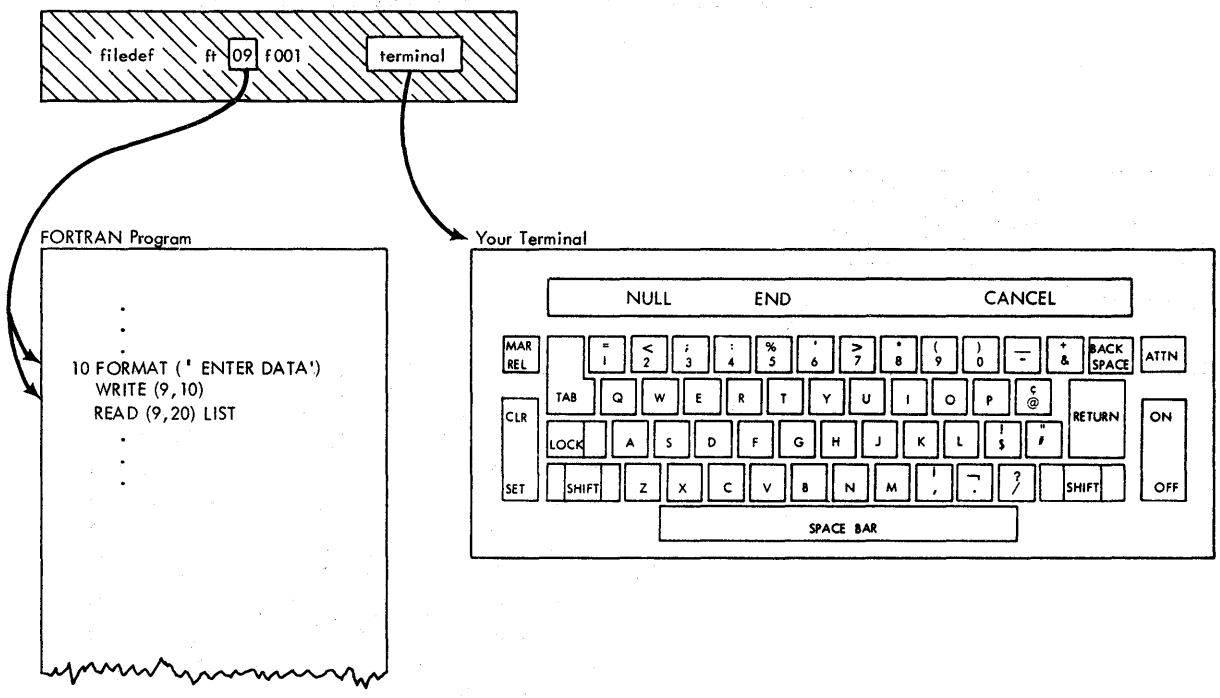


Figure 19. FILEDEF Command for User-defined Terminal Files

For input operations the system waits until you enter data at your terminal. This means that your programs must provide a means for notifying you when to enter data. See the section "Predefined Terminal Input Files" for information on self-prompting. For output operations, data is written at your terminal.

User-defined Punched Card Input Files

Punched card files are associated with your program in one of two ways depending upon the number of card decks to be read.

- Reading One Card Deck

  If your program is going to read only one card deck, that deck is associated with your program through the ddname and device specification of READER in the FILEDEF command that defines it. To define only one punched card input file, you must issue a FILEDEF command with the following format:

  FILEDEF FT*xx* F001 READER [*options*]

where:

xx - is any FORTRAN data set reference number acceptable to your system. Be sure that you do not assign the same data set reference number to both the card reader and punch in the same program.

options - are any valid FILEDEF options for punched card files.

Figure 20 illustrates how the FILEDEF command associates one punched card input file with the input statements in your FORTRAN program.

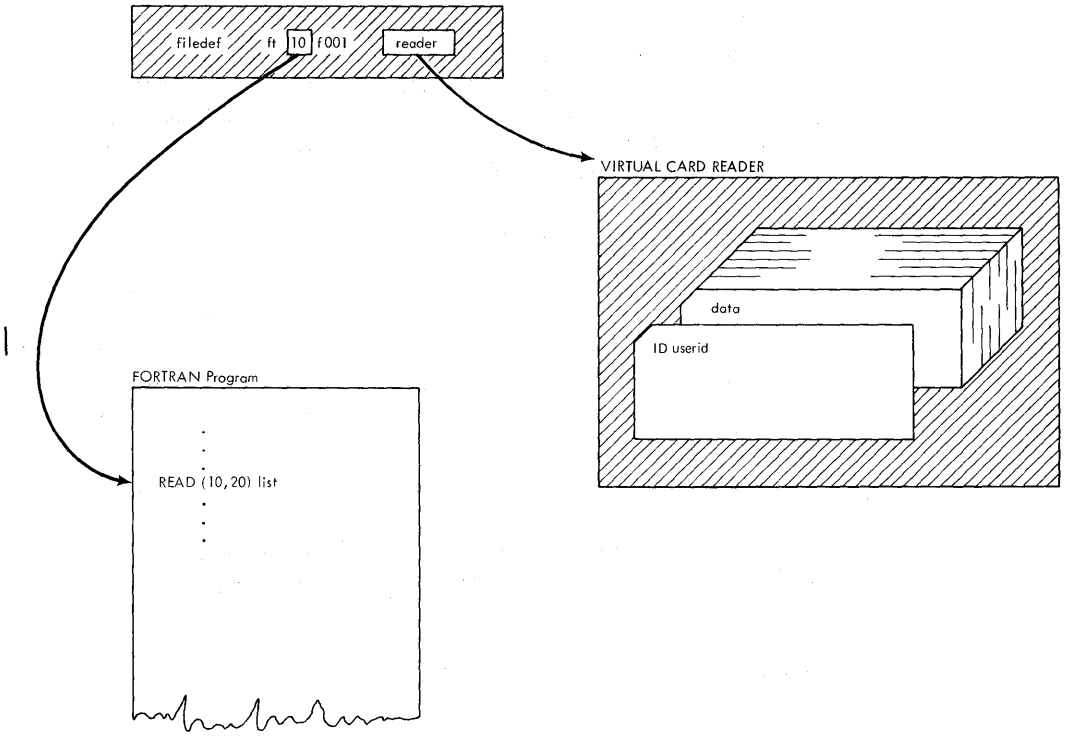
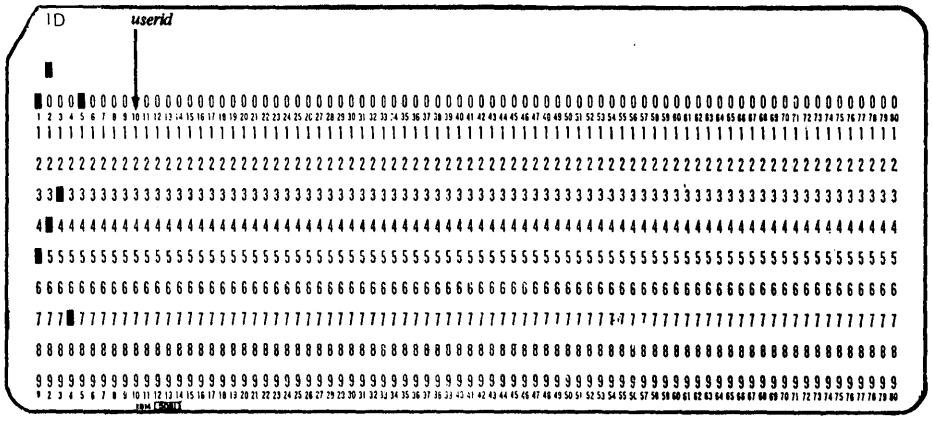


Figure 20. FILEDEF Command for One User-defined Punched Card File

For input operations, you must send the card deck you want read to your computing center or enter it through a remote entry system (if available) before you attempt to execute your program. The system operator will read the card deck into a CMS reader file. Your input statement will read data from this spooled reader file and not from the actual cards. The deck of cards must be identified as follows:



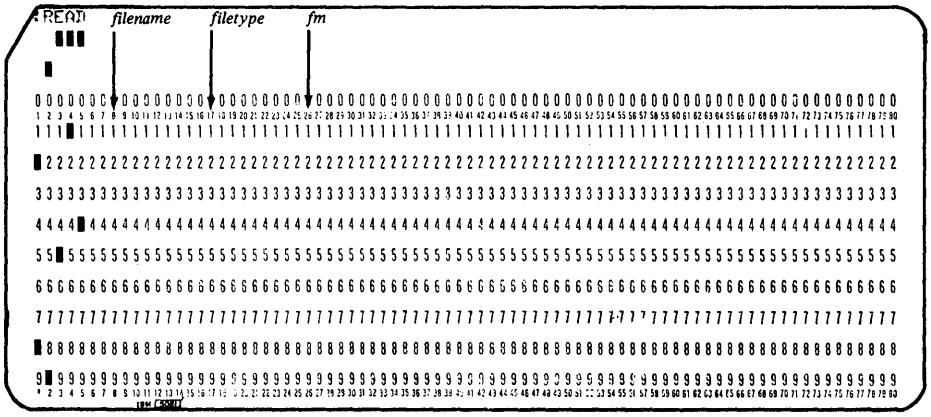
where:

ID must appear beginning in column 1.

userid must appear beginning in column 10.

- Reading More than One Card Deck

  If your program is going to read more than one deck of cards they must be converted into disk files with a READCARD command before they can be associated with your program. Each card deck to be read must be preceded by a :READ card with the following format:



where:

:READ - must appear beginning in column 1

filename - is any valid CMS filename. It must begin in column 8.

filetype - is any valid CMS filetype. It is recommended that you use the ddname for the filetype. The ddname is the default filetype for FORTRAN execution-time files and will remind you what data set reference number has been defined for the file. It must begin in column 17

fm - is optional, but may specify any user disk (A through G). These disks may be any available mode on input (1 through 5). If no mode is specified, A1 is assumed. Files that contain spanned records (that is, RECFM is VS or VBS) must have a mode of 4. If specified, the filemode must begin in column 26.

Multiple punched card files are associated with your program through the ddname, device specification of DISK, and file identifier that you specified in the FILEDEF commands that define them. To define more than one card file you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx* F*yyy* DISK *filename filetype [filemode] [options]*

where:

xx - is any FORTRAN data set reference number acceptable to your system.

yyy - is any valid sequence number, 001 if you are not using multiple files or any number form 001 to 999 if you are using multiple files. (See the section "Using Multiple Files" for more information.)

filename - is the same filename that you specified on the :READ card for this file. If you omit the filename, the system assumes FILE.

filetype - is the same filetype that you specified on the :READ card for this file. If you omit the filetype, the system assumes the ddname of FT*xx* F*yyy*.

filemode - is optional, but if specified is the same filemode that you specified on the :READ card. If you omit the filemode, the system assumes A1.

options - are any valid FILEDEF options for punched card files.

Figure 21 illustrates how the READCARD command converts each punched card deck into a disk file, and how the FILEDEF command associates the disk file thus created with the input statements in your FORTRAN program.

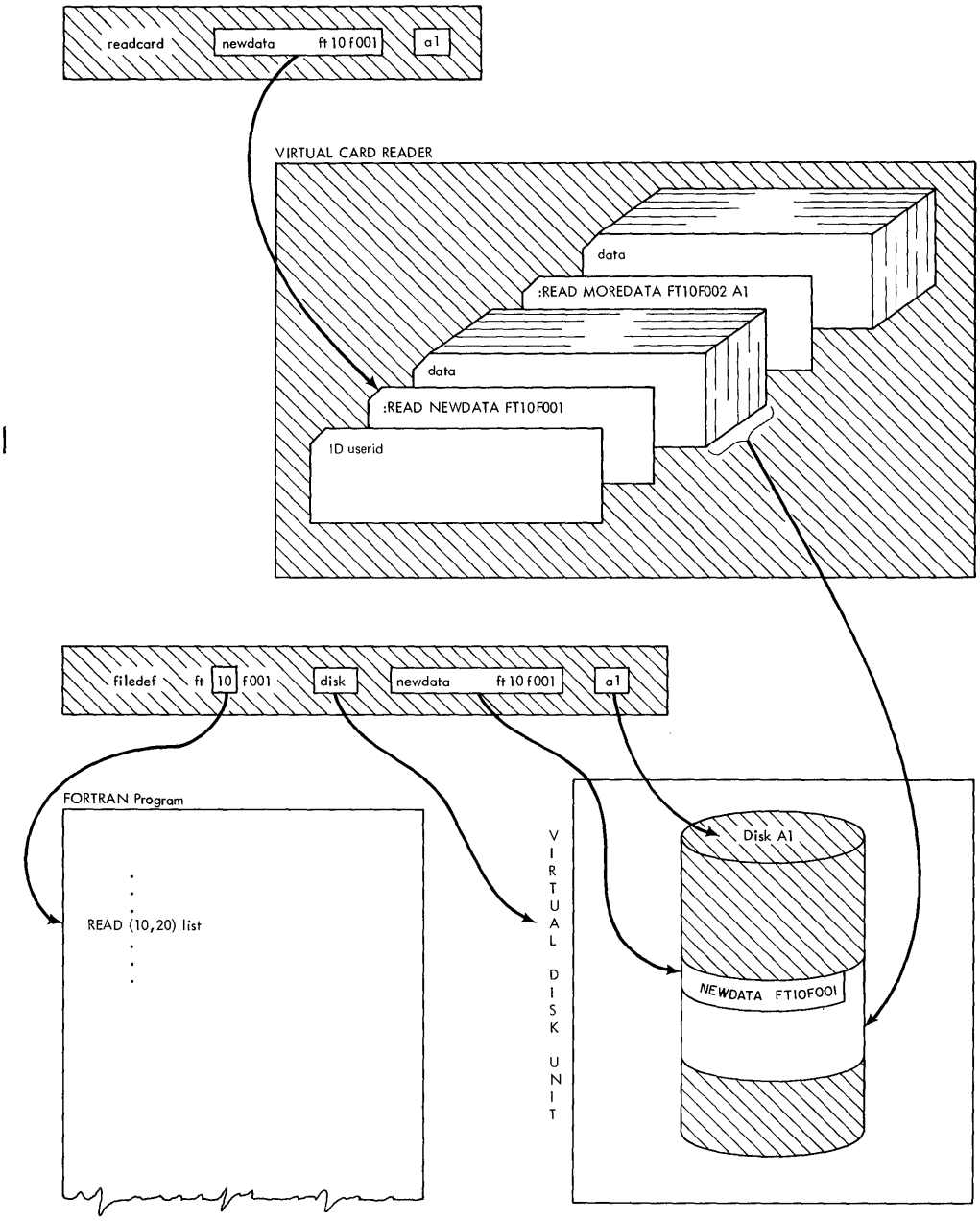


Figure 21. FILEDEF Command for Mutliple User-defined Punched Card Files

For input operations, you must send your card decks you want read (with the appropriate :READ cards) to your computing center or enter them through a remote entry system (if available) before you attempt to execute your program. The system operator will read the card decks into a CMS spooled reader file. The entire set of card decks must be identified with the same header card described previously for only one card deck. When you are ready to run your program, enter a READCARD command for each card deck to be read. The READCARD command has the following format:

READCARD *filename filetype* [*filemode*]

where:

filename - is the same filename that you specified on the :READ card.

filetype - is the same filetype that you specified on the :READ card.

filemode - is optional, but if specified it is the same filemode that you specified on the :READ card.

Each :READ card will transfer the appropriate deck of cards to a disk file with the file identifier that you specified on the :READ card. Once converted, the deck of cards is processed as a disk file.

User-defined Punched Card Output Files

Punched card output files are associated with your program through the ddname and device specification of PUNCH in the FILEDEF command that defines them. To define your own punched card output files, you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx*F001 PUNCH [*options*]

where:

xx - is a FORTRAN data set reference number acceptable to your system. Be sure that you do not assign the same data set reference number to both the card reader and punch in the same program.

options - are any valid FILEDEF options for punched card files.

Figure 22 illustrates how the FILEDEF command associates punched card output files with the output statements in your FORTRAN program.

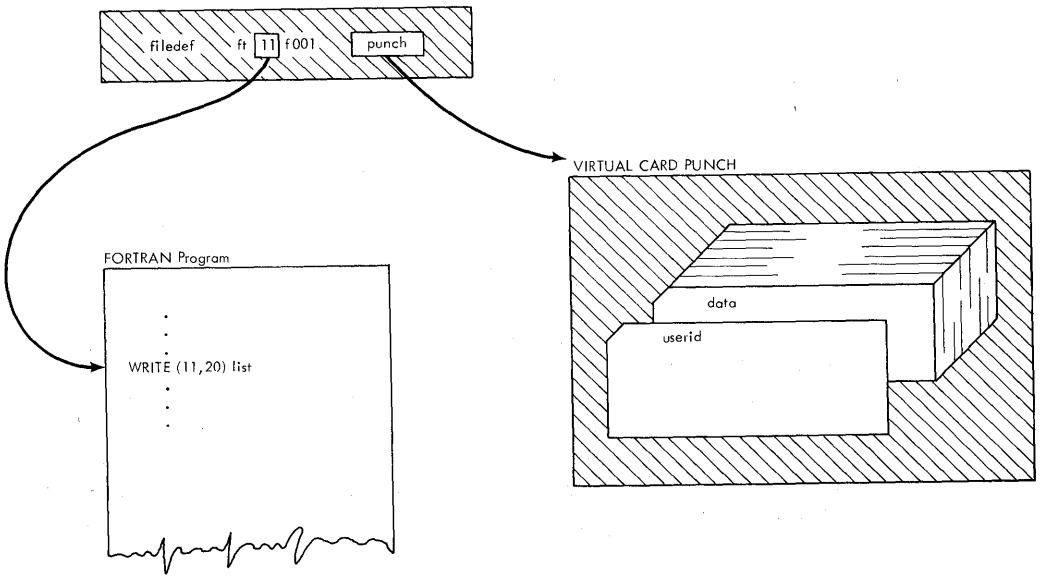


Figure 22. FILEDEF Command for User-defined Punched Card Files

User-defined Printed Output Files

Each printed output file is associated with your program through the ddname and device specification of PRINTER in the FILEDEF command that defines them. To define your own printer files, you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx* F001 PRINTER [*options*]

where:

xx -is any FORTRAN data set reference number acceptable to your system.

options -are any valid FILEDEF options for printed files.

Figure 23 illustrates how the FILEDEF command associates printed files with the output statements in your FORTRAN program.

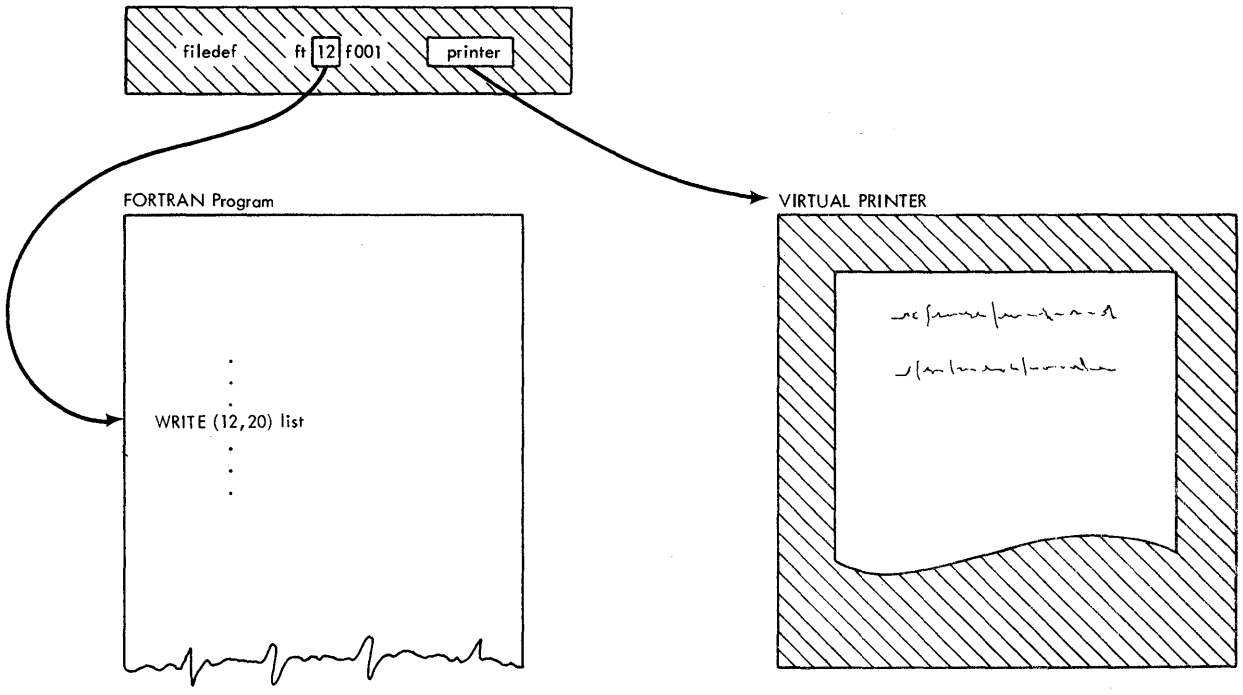


Figure 23. FILEDEF Command for User-defined Printed Files

For output operations, the data you want written is placed by the system in a spooled printer file from which the actual printing will be done. Your WRITE statements do not directly control the printer in the computing center.

Direct Access Files

Each direct access file, whether it is used as input or output, for a FORTRAN object program, requires a file identifier with the following format:

filename filetype [*filemode*]

where:

filename - is any valid CMS filename.

filetype - is any valid CMS filetype. It is recommended that you use the ddname in the FILEDEF for the filetype. The ddname is the default filetype for FORTRAN execution-time files and will remind you what data set reference number has been defined for the file.

filemode - is optional, but may specify any user disk (A through G). These disks may be in any available mode (1 through 5) on input; however, they can only be in modes 1, 4, or 5 on output. If no mode is specified A1 is assumed.

Direct access disk files are associated with your program through the ddname, device specification of DISK, and the file identifier that you specified in the FILEDEF command that defines them. To define your own direct access files, you must issue, for each file, a FILEDEF command with the following format:

FILEDEF FT*xx* F001 DISK *filename type [filemode] [options]*

xx - is any FORTRAN data set reference number acceptable to your system.

filename - is the name of the file to be used or created. If you omit the filename, the system assumes FILE.

filetype - is the type of the file to be used or created. If you omit the filetype, the system assumes the ddname of FTxxF001.

filemode - is optional, but if specified, is the filemode of the file to be used or created. If you omit the filemode, the system assumes A1.

options - are any valid FILEDEF options for direct access files.

Figure 24 illustrates how the FILEDEF command associates direct access files with the input and output statements in your FORTRAN program.

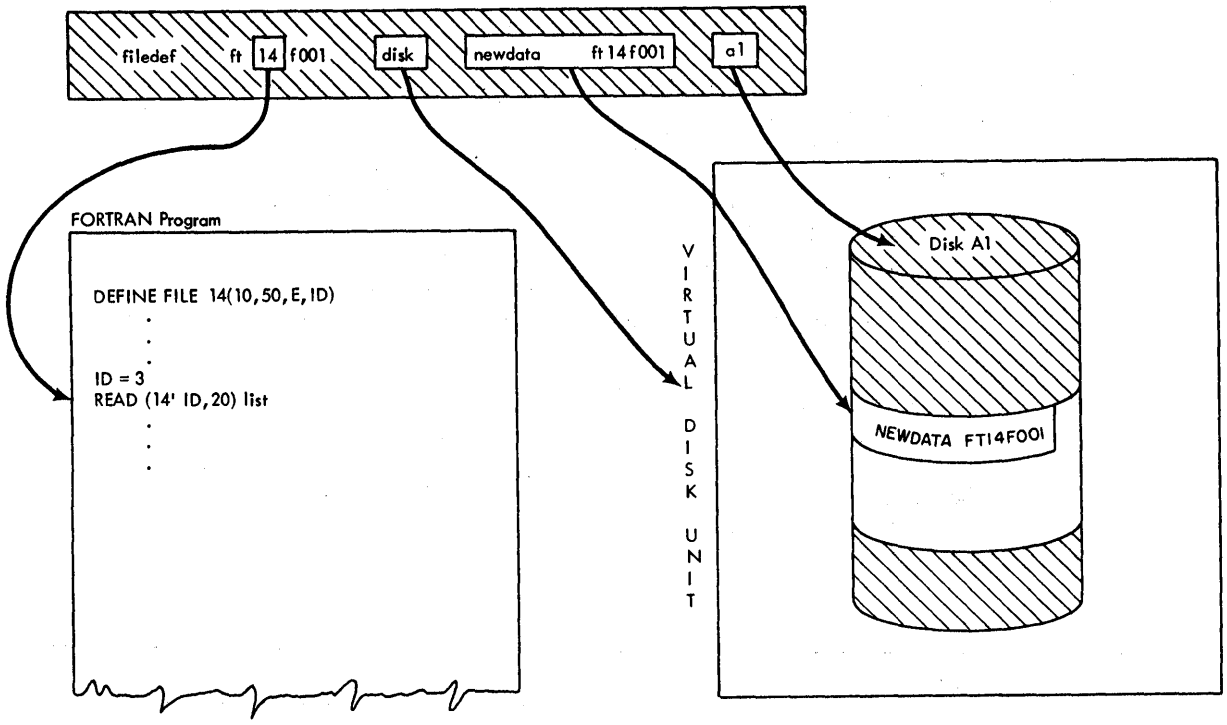


Figure 24. FILEDEF Command for User-defined Direct Access Files

For input and output operations, the system searches for the file identifier specified. If the file exists, data will be read or written from the point specified in the associated variable of the corresponding DEFINE FILE statement. If the file cannot be found, the system will create a file with the file identifier specified in your FILEDEF command and fill it with blanks.

You must be certain that the FORTRAN DEFINE FILE statement accurately describes the file to be used. In addition, you must provide some mechanism within your program to specify the relative record number for multiple files created under the same data set reference number. The number of files specified in the DEFINE FILE statement should be realistic for your needs. For files being created, the number of records that you specify will be blanked out on the disk before actual data is read into it. If you have specified an unnecessarily large number, disk space will be wasted.

CMS supports the FIND statements, although its use slows down the execution of a FORTRAN TEXT file in a time-sharing environment. Input and output overlap is achieved through the sharing of CPU time among the virtual machines that are operating at one time.

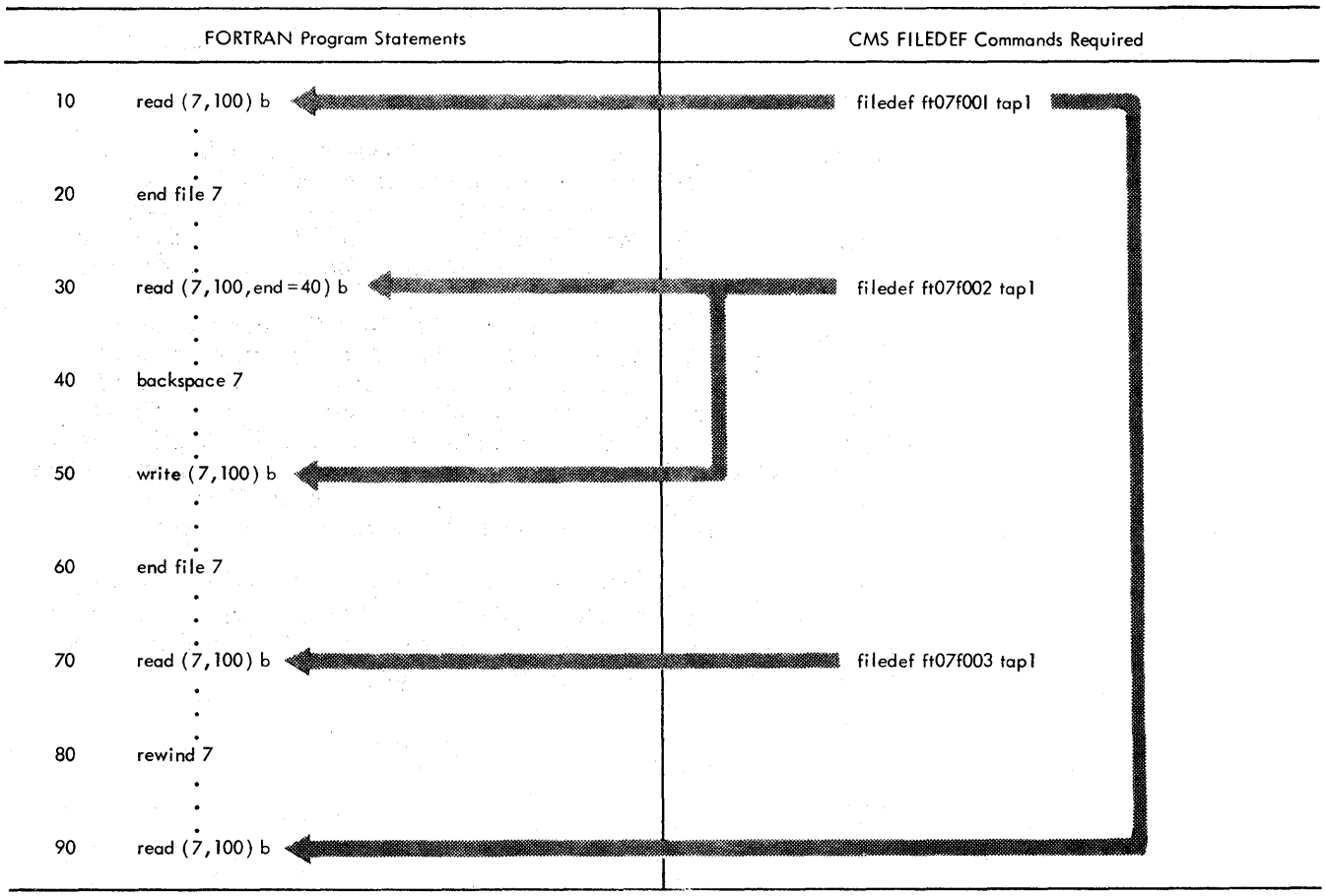
Using Disk and Tape Multifiles

It is possible to create sequential disk and tape multifiles under CMS. A multifile contains several CMS files that are created under the same FORTRAN data set reference number by the same program. The use of multifiles involves the sequence number in the ddname of the FILEDEF command. Sequence numbers are associated with sets of input and output statements in your program depending upon their position and how they are used.

The first set of input or output statements for a specific data set reference number refers to the first file in the multifile. The sequence number for this file is 001. When the end of the first file is reached (either because of an END FILE statement or an END= parameter) the next set of input or output statements for the same data set reference number refers to the second file, which has a sequence number of 002. When the end of the second file is reached, the third file is processed, and so on until the last file has been processed. A FILEDEF command is required for each file (that is, sequence number) in the multifile. The REWIND statement "repositions" the sequence number back to 001.

The BACKSPACE statement can be used to extend a file, that is, add additional data. With a multifile, an end of file condition followed by a BACKSPACE does not position the sequence numbers to refer to the next file. The last file processed remains available. The BACKSPACE statement should not be used with list-directed input and output statements.

Example:



At the beginning of the program the sequence number is 001. Statement 10 reads the first file in the multifile, FT07F001. Statement 20 sets the sequence number to 002, and statement 30 reads the second file, FT07F002, setting the sequence number to 003. When the END= condition is reached statement 40 resets the sequence number back to 002, and statement 50 adds data to the file. Statement 60 sets the sequence number to 003 again. Statement 70 reads the third file, FT07F003. Statement 80 resets the sequence number back to 001, and statement 90 rereads the first file.

Data can be read or written in the first file (sequence number 001) simply by supplying a FILEDEF command for the first file. Care should be exercised in adding new data to an existing file. The new data is placed at the end of the file and will cause the succeeding files to be written over or lost. The procedure for reading or writing into a file with a sequence number greater than 001 is more complex. You must do the following:

1 Perform some input or output operation and include an END FILE statement for each file preceding the one you want.

OR

Read each preceding file supplying an END= parameter that points to the next read.

2 Read or write into the file you want. The same caution about adding data to the first file is applicable here for the following files.

Note: If you execute a REWIND statement before the end of the file is reached, and data that is written into the first file (001) will overlay the data already there. (There is no way to write over the data in a single file with a sequence number greater than 001. All new data added to it is automatically placed after any existing data.) Writing over the data in the first file will not shorten it, if the new data is not as long as the original data. It may, however, lengthen the file, thus eliminating all the original information. Before you write over all the information in the file, first erase the previous information with the ERASE command, since old data that is not written over remains untouched.

FORTRAN IV Programming Considerations

FORTRAN Coding Techniques for Greater Efficiency

In FORTRAN, as with programming in general, there are usually several ways to approach a problem and code a program. However, not all the methods will produce equally efficient or accurate results. Of the techniques listed below, all have been included because they take advantage of the compilers capabilities and produce efficient and accurate executable code. The FORTRAN statements and topics are presented in alphatetical order. Additional information about the FORTRAN IV language is available in the publication *IBM System/360 and System/370 FORTRAN IV Language,* Order No. GC28-6515.

Language Considerations for the FORTRAN IV (G1), Code and Go FORTRAN IV, and FORTRAN (H Extended) Compilers

Arithmetic IF Statements

When using arithmetic IF statements, avoid making tests that depend on the real floating-point zero. Many real numbers must be represented approximately (although to a high degree of accuracy) in the internal code of the computer. Slight errors resulting from computation with these numbers may prevent an anticipated value of zero from being obtained, and, hence invalidate any test for zero.

A fixed-point overflow condition in an arithmetic IF statement results in the following action for G1 and Code and Go:

- If the integer is positive, a negative branch is taken, that is, the first branch.
- If the integer is negative, a positive branch is taken, that is, the third branch.

For H Extended, if the integer equals zero the second branch is taken.

BACKSPACE Statement

The BACKSPACE statement can be used to extend a file, that is, add additional data to it. With a multifile, an end of file condition followed by a BACKSPACE statement does not position the sequence number to point to the next file. The last file processed remains available. The BACKSPACE statement should not be used with the LIST-directed input and output statements.

FIND Statement

CMS supports the FIND statement, although its use slows down the execution of a FORTRAN TEXT file in a time-sharing environment. Input and output overlap is achieved through the sharing of CPU time among the virtual machines that are operating at one time.

List-Directed Input and Output

The following rules affect the use of list-directed input and output:

- List-directed output statements can be used to create FORTRAN data files that are acceptable as input to PL/I processors, provided that these data files do not contain COMPLEX data types.
- The block size must be large enough to contain the largest data item other than a complex number. For a complex number, the block size should be larger than half the length of the item plus a comma. If the block size is not large enough the remainder of the input or output list is ignored.

For a complete description of list-directed input and output see the section "Library Features Available with the FORTRAN IV Mod I and Mod II Libraries."

Literals in Data Initialization

In initializing an array, you should consider the following:

- Any element of an array may be initialized by subscripting the array name. Only one element is initialized; if excess characters are specified, they are not placed into the next element. An array element that is partially filled is padded on the right with blanks. The example below illustrates how individual array elements are initialized.

  Example:

```
DIMENSION A(10)
DATA A(1),A(2),A(4),A(5)/'ABCD','QRSTUVW','123','666666'/
```

  The array elements contain the following:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
ABCD	QRST		123	6666					

- Several consecutive elements of an array may be initialized with a single literal constant by specifying the array name without a subscript. Data spill occurs over as many elements as are necessary to insert the entire constant (as long as the constant does not exceed the limits of the array). The example below illustrates how several array elements may be initialized with a single literal constant.

  Example:

```
DIMENSION ARRAY(9)
DATA ARRAY/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
```

  The array elements contain the following:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)
ABCD	EFGH	IJKL	MNOP	QRST	UVWX	YZ		

  Using this method, initialization always begins with the first element of the array. To begin initialization with an element other than the first you can use an EQUIVALENCE statement.

  Example:

```
DIMENSION ARRAYA(10), ARRAYB(5)
EQUIVALENCE(ARRAYA(6),ARRAYB(1))
DATA ARRAYB/'ABCDEFGHIJKLMNOPQRST'/
```

  The arrays will be initialized as follows:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
					B(1)	B(2)	B(3)	B(4)	B(5)
					ABCD	EFGH	IJKL	MNOP	QRST

- Individual elements of an array may be initialized after initializing several elements with an unsubscripted array name. Each constant must follow the array name that is to be initialized. The example

below shows how the two methods may be combined into one operation.

Example:

```
DIMENSION ARRAY(5)
DATA ARRAY/'ABCDEFGH'/,ARRAY(4)/'4444'/,ARRAY(5)/'5555'/
```

The array will be initialized as follows:

A(1)	A(2)	A(3)	A(4)	A(5)
ABCD	EFGH		4444	5555

If each constant that is to initialize a part of the array is not specified immediately after the elements that will contain them, data that has overflowed into subsequent array elements may be replaced by data that was incorrectly specified. The example below illustrates one possible unintentional result.

Example:

```
DIMENSION A(3)
DATA A,X/'ABCDEFGHIJKL',10.0/
```

The array will be initialized as follows:

A(1) A(2)	A(3)	X
ABCD 10.0	IJKL	

As you can see the compiler assumes that the second constant is intended for the second array element. The correct way to code this array is shown in the example below.

Example:

```
DIMENSION A(3)
DATA A/'ABCDEFGHIJKL'/,X/10.0/
```

This would result in the following initialization:

A(1)	A(2)	A(3)	X
ABCD	EFGH	IJKL	10.0

The FORTRAN IV G1, Code and Go FORTRAN, and H Extended compiler will flag truncation and spill in data initialization as an error; however, executing the TEXT file will produce the expected results described above.

Logical IF Statement

Use of the logical IF statement rather than a comparable arithmetic IF statement can result in a more efficient compilation. Statement 5 below is more efficient than statement 6.

Example:

```
  5   IF (A.GT.B) GO TO 20
  6   IF(A-B) 10,10,20
 20   CONTINUE
```

Each set of logical comparisons occurring in a logical IF statement is analyzed separately by the compiler.

Example:

```
(A.LT.B.OR.C.GT.F.OR. NOT.L) GO TO 10
```

This statement is analyzed as though it were written:

```
IF  (A.LT.B)  GO  TO  10
IF (C.GT.F) GO TO 10
IF ( .NOT.L) GO TO 10
```

Therefore, if A is less than B, the remainder of the statements are not evaluated. You can affect the efficiency of your execution by the order in which you specify the logical comparisons. For example, if you expect C to be greater than F more often than A is less than B, test for C being greater than F first.

PAUSE n Statement

The PAUSE statement message or number will be displayed at your terminal and execution of your program halted. To restart your program, hit the ATTN key.

READ Statement

The ERR= parameter in the READ statement causes a branch to another statement if an input or output error is encountered. The READ statement that encountered the error does not make the data available to your program. A second READ statement is necessary to do this. Thus, you can direct the ERR= parameter to an error processing routine that rereads in the error and disposes of it before returning to normal processing.

Example:

```
  5   READ (8,100,ERR=200)A
100   FORMAT(I10)
      GO TO 300
200   READ(8,100)AX
300   CONTINUE
```

If the ERR= parameter is not included, an input or output error causes the program to stop processing.

RETURN Statement

The RETURN statement is used in subprograms to return control to the program that called it. If this statement is included in a main program, it returns control to the operating system. As a note to assembler programmers, the RETURN statement issues the following codes in register 15:

0	If the RETURN statement was executed in a normal fashion in either a main program or subprogram.
4*i	If a RETURN i statement was executed in a subprogram.
16	If a terminal error was detected during execution in a library subprogram.

STOP n Statement

The number specified in the the STOP *n* statement should not be larger than 4095, since a large number causes an overflow into a system return code field. The return code that will be issued is *n* modulo 4096, that is, the remainder after dividing *n* by 4096. However, the number you specified will be displayed at your terminal. To restart processing, hit the ATTN key.

Unformatted Forms of Input and Output Statements (Not Including List-Directed)

The unformatted form of an input or output statement results in a faster data transfer rate into and out of storage, since no data conversions are performed. When operations are being performed on intermediate data files, (those which are used internally by the program and which you never see) the use of unformatted data increases program efficiency. In the example below, statement 11 is more efficient than statement 10.

Example:

```
      DIMENSION A(100)
   10 WRITE(20,9)A,B
    9 FORMAT(100E13.3)
   11 WRITE(20)A,B
```

Language Considerations for the FORTRAN IV (G1) and Code and Go FORTRAN IV Compilers Only

Array Notation in Input and Output Statements

The use of array notation is more efficient than an implied DO. In the example below, statement 4 is more efficient than statement 3.

Example:

```
3 READ(9)(A(I),I=1,10)
4 READ(9)A
```

Language Considerations for the Code and Go FORTRAN Compiler Only

Free-Form Input

The following rules govern the use of free-form input format:

- *Maximum statement length:* 1320 characters, excluding statement number and statement break characters.
- *Maximum Line Length:* 81 bytes including statement numbers and statement break characters.
- *First line of statement:* This may start in any typing position.
- *Statement numbers:* The first line of a statement may contain, as the first non-blank characters of that line, a statement number consisting of from one through five decimal digits. Blanks and leading zeros in a statement number are ignored as are any blanks preceding the statement number. A blank need not separate a statement number from the first nonblank character that follows the statement number.
- *Continued line:* A line of a statement to be continued is indicated by terminating the line with a hyphen.
- *Continuation line:* A line following a continued line. A continuation

line can begin in any typing position except where a literal constant is being continued, in which case the line must begin in position 1. A continuation line may also be continued; up to 19 continuation lines are permitted in a single statement.

- *Comment line:* Any noncontinuation line with an asterisk (*) or a double quote (") as its first non-blank character. A comment line cannot be continued, but multiple comment lines may be used.

Note: The default line escape character under CMS is also a double quote ("). You must change this escape character to a character other than a double quote before entering free-form source comments.

- *End line:* An end line consists of the characters END preceded by, interspersed with, or followed by a maximum of 63 blanks (that is, it may not be continued on a subsequent line).

The standard form statements:

```
Typing Position:     1         7
                     c         sample text
                           10  d=10.5
                               go to 56
                          150  a=b+c*(d+e**f+
                               cg+h-2.*(g+p))
                               c=3.
```

could be written in free form as the following:

```
Typing Position:     1       7
                     '' sample text
                     10 D=10.5
                     go to 56
                     150 a=b+c*(d+e**f+-
                     g+h-2.*(g+p))
                     c=3/
```

A sift utility is provided with the Code and Go processor that will produce fixed-length standard-form FORTRAN input records from free-form statements. Fixed-length records may be submitted to other compilers for processing. You can invoke the sift utility with the CONVERT command. The GOFORT compiler command allows you to specify whether your source statements are fixed- or free-form. See Appendix D for more detailed information on using the sift utility.

Language Considerations for the FORTRAN IV (H Extended) Compiler Only

Array Notation in I/O Statements

Array notation is the preferred method for coding I/O lists. Under all levels of the OPTIMIZE option, whenever possible, implied DO statements are

treated as arrays. In the example, below, statement 4 implementation would be substituted for statement 3:

```
      DIMENSION A(100)
3     READ (9) X(A(I),I=1,100)
4     READ (9) A
```

for a nest of implied DOs, array notation is implemented if the following conditions are met:

1. Only one list item is included in the range of any of the implied DO levels; this list item is not a DO variable.
2. A list item which is an array does not have variable dimensions.
3. The initial, test, or increment value of an inner DO loop is not the DO variable of any outer DO loop.
4. The DO variable does not occur as a subscript for an element of the subscript.
5. The DO variable does not occur as a subscript for an element of the subscript.
6. An array subscript contains only constants or variables as operands.
7. Each arithmetic term occurring in an expression of an array subscript meets all the following conditions if that term contains a DO variable:
   a. No exponentiation or division occurs.
   b. All operands except the DO variable are integer constants.
   c. No DO variable occurs more than once.
8. The maximum iteration for each DO level of an implied DO with constant DO limits is 2^{24} (16,777,216) times.

BASE Registers

Register 13 is the primary base register. The following must be addressable using register 13: 18 words for the save area, 1 word for the adcon for register 12, branch tables for all computed GOTOs, and parameter lists for all call statements and external functions. Register 12 is the secondary base register. If register 13 does not reach the end of the parameter lists or if register 12 and register 13 are both exceeded, a level 16 error is issued and the compilation is deleted. This may occur in a program with many branch tables or parameter lists.

EQUIVALENCE Statement

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

Example:

```
EQUIVALENCE (VARA,ARAYA(3),ARAYB(5),ARAYC(10))
```

This statement would be compiled faster by reversing the order.

Example:

```
EQUIVALENCE (ARAYC(10),ARAYB(5),ARAYA(3),VARA)
```

To reduce compilation time and save internal table space, equivalence groups should be combined where possible.

Example:

```
EQUIVALENCE (ARRA(10,10),VAR1),(ARRB(5,5),VAR1)
```

This statement could be recoded more efficiently.

Example:

```
EQUIVALENCE (ARRA(10,10),ARRB(5,5),VAR1)
```

EXTERNAL Statement

By placing an ampersand before a function name in an EXTERNAL statement, the programmer "detaches" that name, that is, declares it to be the name of a user-supplied function even though the name may be the same as a function or subroutine appearing in the FORTRAN IV Library (Mod II). If the function name following the ampersand is not the same as a library function, it is still considered detached; no diagnostic action is taken.

Also, by specifically typing a subprogram name, the programmer detaches the name from the library.

Example:

```
REAL*8 SIN
```

SIN will be automatically detached from the library.

GENERIC Statement

The GENERIC statement requests the use of the Automatic Function Selection facility of the FORTRAN IV Library (Mod II). As a result, the appearance of the generic name in a program causes the appropriate function name to be substituted according to the length and type of the arguments specified. For example, the generic name COS, specified with arguments of REAL*8 causes the function DCOS to be substituted.

To avoid conflict with specific references to functions, the function names substituted as a result of automatic function selection are aliases, which you cannot otherwise specify. Aliases beginning with the characters IH$$ refer to function names three characters in length, and IH$ to names four to six characters in length. Names six characters in length are automatically reduced to five characters by deleting the next to last characters before prefixing the name with IH$. For example, the function DCOTAN substituted for COTAN would appear to have the name IH$DCOTN.

Name Handling

The compiler places names for variables, arrays, and subprograms into a table and searches the table whenever a reference is made to a name. The table is divided into six strings. Names that are one character long are placed into the second string; and so on. For faster compiling, allocate names as evenly as possible among the sizes.

OPTIMIZE Compiler Option

The OPTIMIZE option of the FORTHX compiler command improves execution-time and reduces the amount of the executable code produced.

OPTIMIZE(1) causes the entire program to be treated as a loop, with individual sections of coding, headed and terminated by labeled statements, treated as blocks. The executable code is made more efficient by:

- Improving local register assignment. (Variables that are defined and used in a block are retained where possible in registers during the processing of the block. Time is saved because the number of load and store instructions are reduced.)
- 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 are reduced.)
- Improving branching by the use of RX format branch instructions in the executable code. (An RX branch instruction saves a load instruction and reduces the number of required address constants.)

OPTIMIZE(2) performs optimization beyond that performed with OPTIMIZE(1) by:

- Assigning registers across a loop to the most active variables, constants, and base addresses within the loop.
- Moving outside the loop many computations which need not be within the loop.
- Recognizing and replacing redundant computations.
- Replacing, where possible, multiplication of induction variables by addition of those variables. (An induction variable is one that is only incremented by a constant or a variable whose value remains constant in the loop.)

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

Registers 0, 1, and 12-15 are required by the system. The remaining registers, 2-11, are available for use by optimization techniques. Branch optimization reserves registers 11, 10, and 9, if needed and as needed, for object program instructions in large source programs.

Programming Considerations When Using OPTIMIZE(1) and OPTIMIZE(2)

Although these options can result in more efficient code, they place additional responsibilities on you the programmer and require additional programming considerations.

Using COMMON Statements: Variables in COMMON are normally not stored unless an input/output statement or a subroutine call using them is issued.

Using Subprograms: If a user-defined subprogram is given the same name as a FORTRAN-supplied subprogram (for example, SIN, ATAN), errors may be introduced during optimization. To avoid these errors, specify the subprogram name in an EXTERNAL statement (with an ampersand preceding the subprogram name).

If the extended error handling facility is specified and a user-supplied subroutine uses program variables, there is no assurance that correct values will be available.

If a subprogram is called at one entry point for the purpose of initializing arguments and at another entry point for computations, the latter call must include an argument list to ensure that the subprogram will receive current values for the arguments. This rule applies when the subprogram refers to the arguments by name (that is, accesses them in their locations in the calling routine rather than through local variables).

In the following example, the updated value of N will be correctly stored and transmitted to the subprogram. If the call to the subprogram did not include the argument list, N would be updated in a register but not in storage.

Example:

```
        CALL INIT(N)
        .
        .
        .
10      CALL COMP(N)

        N=N+1
        .
        .
        .
        GO TO 10
        SUBROUTINE INIT(/J/)
        .
        .
        .
        ENTRY COMP(/J/)
```

Using COMMON Blocks: Because each COMMON block is an independent program unit, it is independently relocatable, and thus requires a base address that specifies its beginning point in storage. Each base address must be stored into a register in order to be accessible. If many COMMON blocks are defined, the need to load base addresses slows down processing time. If multiple blocks can be combined into one block of less than 4096 bytes in length (the maximum number that can be accommodated in a register) one base register can serve to address each variable.

Using the Assigned GO TO Statement: If the list of statement numbers is incomplete in an assigned GO TO statement, errors that were not present in the unoptimized code may appear. Hence, you should be sure that all GO TO statements have complete lists of statement numbers.

Programming Considerations When Using OPTIMIZE(2)

OPTIMIZE(2) evaluates expressions and eliminates common expressions. For example, if an expression occurs more than once, and the program path always passes through the first occurrence of that expression to reach a second occurrence without changing the value of the expression, the first value of the first occurrence is saved and used instead. This is done for both full expressions and intermediate results within expressions.

Example:

```
A = C + D
      .
      .
      .
F = C + D + E
```

The common expression C + D is saved after its first evaluation in A and is used in F without repeating the computation.

Using Subprograms: If a FORTRAN library function is called, the computational reordering performed during optimization may cause unexpected results.

Example:

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

The optimization technique moves the library function call before statement 9, causing the square root computation ot occur before the test for zero. To avoid this situation, the program should be rewritten as follows:

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

Using the FORTRAN Subroutine Libraries

The FORTRAN IV Library (Mod I) or the FORTRAN IV Library (Mod II) can be used under CMS. The libraries are available in up to three CMS disk files that were determined when the library was installed in your system. These files are searched whenever a reference to a subroutine is encountered after a LOAD or INCLUDE command has been issued. Figure 25 outlines the contents of the library files depending upon the number of files and the availability of the Extended Error Handling Facility.

The files you choose to use must be identified in a GLOBAL command before their contents can be referred to. You can identify them in a PROFILE EXEC procedure or with individual GLOBAL commands prior to executing your FORTRAN programs. See the *VM/370 Command Language User's Guide*, Order No. GC20-1804 for a description of these commands. In addition, you may refer to the publications *IBM System/360 OS FORTRAN IV Mathematical and Service Subprograms*, Order No. GC28-6816 and *IBM System/360 OS FORTRAN IV Mathematical and Service Subprograms Supplement for Mod I and Mod II Libraries*, Order No. SC28-6864 for a complete description of the library routines and their functions.

Before attempting to use any of the FORTRAN libraries, determine from the system administrator in your computing center, the number of library files, the names of the files, and the availability of the Extended Error Handling Facility.

If you have the Code and Go compiler with either the Mod I or Mod II library, you will need to include an entry in your GLOBAL command for the text library TSOLIB, which contains CMS system routines that support the operation of the compiler.

If you have the H Extended compiler and Mod II library and will be using extended precision arithmetic, you will also need to include in your GLOBAL command an entry for the text library CMSLIB, which contains the extended precision simulation routines.

Note: You may substitute a routine of your own for any of the FORTRAN library subprograms and give it the same name as the library subprogram. However, in resolving references, CMS searches through libraries specified in your GLOBAL command. Therefore, care should be exercised in executing programs that use a FORTRAN library subprogram for which you also have a TEXT file with the same filename. If you do not want your TEXT file used, you must rename it prior to executing your program.

Library	Available in					
	One File	Two Files		Three Files		
	File 1 Contains	File 1 Contains	File 2 Contains	File 1 Contains	File 2 Contains	File 3 Contains
Mod I and Mod II with the Extended Error Handling Facility (EEH)	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines • Error Handling Routines with EEH	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines • Error Handling Routines with EEH	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines • Error Handling Routines without EEH	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines	• Error Handling Routines with EEH	• Error Handling Routines without EEH
Mod I and Mod II without the Extended Error Handling Facility (EEH)	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines • Error Handling Routines without EEH	• Initialization and Termination Routines • Math and Service Routines • I/O Routines • Conversion Routines	• Error Handling Routines without EEH			

Figure 25. Contents of the FORTRAN Libraries (Mod I) and (Mod II)

Library Features Available With the FORTRAN IV Mod I and Mod II Libraries

The following features are supported by the G1, Code and Go, and H Extended compilers.

List-directed and Formatted Input/Output

If your installation has the FORTRAN IV Library (Mod I) or (Mod II) you can use the list-directed input and output (often called "list I/O") facilities which are provided by the interface between the FORTRAN IV Compilers and the Libraries. List-directed input and output is simpler to use than formatted FORTRAN I/O and is particularly useful for terminal input and output.

With formatted I/O you must write a FORMAT statement to specify to the compiler the format of your input and output data. When you list I/O statements refer to the terminal, the compiler prompts you for data and then recognizes the format of each data item as you enter it from the terminal. It also creates its own format for output data to the terminal.

To enter data using list-directed I/O, make a list of data items in the sequence in which they are to be read. Separate individual items by commas, blanks, tabs, or carrier returns. Write individual entries in real, integer, complex, or double-precision formats. When you wish to use list-directed input/output, omit the FORMAT statement, and replace the FORMAT statement number in your READ or WRITE statement with an asterisk. The data set reference number remains the same and represents whatever device your installation has assigned to that number.

For example, suppose you have a program that reads in five input values for variables A, B, C, D, and E. You may now supply the values using the terminal as your input device. Assume that 5 is the data set reference number for reading data from the terminal. Code a READ statement as follows:

for list-directed I/O

```
50 read (5,*) a,b,c,d,e
```

for formatted I/O

```
50 read (5,10) a,b,c,d,e
```

Now execute your program. In both cases, your program will execute up to statement 50, the READ statement. The program now needs input data from you. The values to be supplied are 2.3, 6.74 2.1E+7, 6.4E03, 0.44432. In the list-directed I/O case, the following happens:

The system prompts you for the data with a question mark and the READ statement number. The interchange between you and the system is:

```
terminal:  ? 00050
you:       2.3, 6.74 2.1e+7
           6.4e-03
           9.44432
```

You could just as well have strung the five values out on one line, all separated by commas, or all separated by one or more spaces, or you could have entered each value on a separate line.

If you decide to omit an input value (leaving a variable unchanged from its previous value) use successive commas. In the above example, if you wanted to leave c without a new input value, you might enter:

```
2.3,6.74,,6.4e-03,9.44432
```

If the value or values you are leaving out are the last ones in the list, end the list with a slash. For example, to read in just A,B and C, you would enter

```
2.3 6.74,2.1e+7/
```

In the formatted case, you would have to write a FORMAT statement similar to:

```
10 format(2f3.2,2e10.1,f7.5)
```

No prompting is provided for this data. The terminal keyboard unlocks, and you can type in the data required. The data must be typed in exactly as it would be punched on a card:

```
230674000002.1e70000.64e-30944432
```

Since no prompting is provided, you should make a list of your data in the order and form that it is needed, so that you can enter it correctly. An alternative and recommended method of entering data is to issue prompts yourself: insert a WRITE before each READ, with a literal message to yourself to enter the data. In the above example, the statement immediately preceding statement 50 might be:

```
write(6,5)
```

where statement 5 is:

```
5 format('enterdatafora,b,c,d,e,')
```

Note that the WRITE statement uses 6 as the data set reference number where the READ statement used 5. Even though the terminal is the I/O device in both cases, FORTRAN requires different data set reference numbers for reading and for writing. Data set reference numbers 5 and 6 are the defaults used in this book for terminal input/output. Check to be sure what the terminal data set reference numbers for read and write are at your own installation.

You can mix conventional and list-directed I/O in your program. This may be necessary since list-directed I/O cannot handle the writing of literal data.

The following is an example of mixed list-directed and formatted I/O as it might occur in a simple program to average three numbers.

Your program reads as follows:

```
 50   read(5,*)x,y,z
      sum=xx+Y+z
      avg=sum/3
      write(6,100)
100   format('theaverageofthethreenumbersis:')
      write(6,*)avg
```

When your program runs, the I/O at your terminal looks like this:

```
system:   ? 00050
you:      70,105,91
system:   the average of the three numbers is:
system:   88.666
```

Extended Error Handling

The extended error handling facility is an optional feature of the FORTRAN libraries that may be incorporated into your installation. If it is, you will have additional diagnostic information available to you for errors occurring during load module execution. Check with your system administrator to see if your installation has extended error handling. If it does not, this section does not apply to you.

When extended error handling is incorporated, it provides you with the following features:

- execution-time error messages that are more precise than those issued with standard execution-time diagnostic facilities.
- the ability to continue execution after an error occurs.
- the opportunity to examine errors and to use either standard or user-supplied corrective action. (If you are a typical CMS user, you will be using the corrective actions that are built into your installation's system. More experienced system programmers can use the methods described later on to modify existing ones.)

Extended error handling is superior to standard error handling in a number of ways. Both types of error handling produce messages identifying any errors that are found when a program is executed, and produce a summary error count at the end of the job indicating the number of times each error occurred. Extended error handling, however, allows your installation to exercise control over the action performed by the system when an error is encountered. With extended error handling, it is possible to direct the system to perform standard corrective action and continue processing, or have the system transfer control to an installation-supplied routine, which can perform any type of corrective action desired. Your installation also has control over

the number of times a particular error is allowed to occur before program termination, the maximum number of times each message may be printed, and whether or not a traceback map is to be printed for the error.

Your installation controls these features by means fo a collection of instructions stored in a list called the "option table". When the FORTRAN IV Library is installed, there are standard IBM-supplied entries in the option table; however, your installation can modify these entries to suit its own needs. These modifications include adding error conditions, changing aspects of handling particular error conditions, or substituting an installation's own corrective routine for the IBM-supplied corrective action for a particular error. The result is a permanent option table tailored to your installation. Occasionally, a programmer may wish to change features of the option table temporarily to suit his own needs. He can do this from the terminal. Changes made this way last only as long as the session in which they are made; after a programmer logs off, the system reverts to the permanently stored option table.

As a CMS user you will normally not be concerned with making changes of any type in the option table. You can find out from your system administrator what provisions for error handling are contained in your installation's option table.

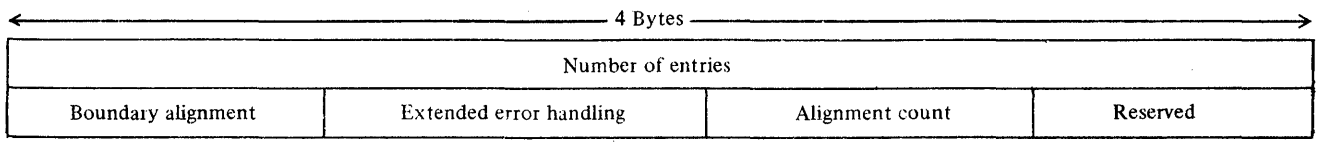
The general form of the option table and the kind of things in it are given in Figures 26, 27 and 28. For the majority of applications, this description together with the error messages produced by the extended error handling facility (presented in the "Diagnostic Messages" section of this book) will be sufficient so that the extended error handling will present no problems for you in your terminal sessions. You need not bother with the sections describing alterations to the option table.

The Option Table

The option table consists of two sections; a preface and a number of entries which describe error conditions. The preface is a doubleword entry describing program handling characteristics, such as whether extended error handling has been specified. Each error condition entry is a doubleword entry describing characteristics of a particular error condition. For example, an error condition entry will define the number of occurrences of the error encountered, or whether an installation-supplied routine is provided for handling the error. IBM provides a standard set of error conditions. Your installation may have additional entries of its own. Figures 26 and 27 describe the fields of the option table and list the values supplied for each entry when your installation is set up. Figure 28 lists the option table defaults values for the available error codes.

Systems programmers who want to make temporary changes to the extended error handling option table can find more detailed information in Appendix D.

Format



Description

Field Contents	Length in Bytes	Field Description
Number of entries	4	Number of entries in the Option Table. The default setting is 95.
Extended error handling	1	Indicates whether extended error handling facility was chosen at installation time. FF(hexadecimal) = EXCLUDE 00(hexadecimal) = INCLUDE The default setting is FF.
Alignment count	1	Maximum number of boundary alignment messages when extended error handling facility is not chosen. The default setting is 10.
Reserved	1	Reserved for future use.

Figure 26. Option Table Preface

Field Contents	Field Length in Bytes	Default[1]	Field Description
Number of error occurrences allowed	1	10[2]	Number of times this error condition should be allowed to occur. When the value of the error count field (below) equals this value, job processing is terminated. The number may range from 0 to 255. A value of 0 means an unlimited number of occurrences[3]. A value greater than 255 sets the field to 0.
Number of messages to print	1	5[4]	The number of times the corresponding error message is to be printed before message printing is suppressed. A value of 0 means no message is to be printed.
Error count	1	0	The number of times this error has occurred. A value of 0 indicates that no occurrences have been encountered.
Option bits	1	42 (hexadecimal)	Eight option bits defined as follows:
User exit	4	1	Indicates where a user corrective routine is located. A value of 1 indicates that no user-written routine is available. A value other than 1 specifies the address of the user-written routine.

Bit	Setting	Default	Explanation
0	0	0	No control character supplied for overflow lines.
	1		Control character supplied to provide single spacing for overflow lines.
1	0		Table entry cannot be modified[5].
	1	1	Table entry can be modified.
2	0	0	Fewer than 256 errors have occured.
	1		More than 256 errors have occured. (Add 256 to error count field above to determine the number.)
3[6]	0	0	Do not print buffer contents with error message.
	1		Print buffer contents.
4	0	0	Reserved.
5	0	0	Unlimited printing of error messages not requested; print only default number of times.
	1		Unlimited printing requested; print for every occurrence of error.
6	0		Do not print traceback map.
	1	1	Print traceback map.
7	0	0	Reserved.

[1] The default values shown apply to all error numbers (including additional user entries) unless excepted by a footnote.

[2] Errors 208, 210, and 215 are set as unlimited, and errors 217 and 230 are set to 1.

[3] An unlimited number of errors may cause the FORTRAN job to loop indefinitely until the operator intervenes.

[4] Error 210 is set to 10, and errors 217 and 230 are set to 1.

[5] The entry for error 230 cannot be modified.

[6] The entry is set to 0 except for errors 212, 215, 218, 222, 223, 224, and 225.

Figure 27. Option Table Entry Format

Error Code	Number of Errors Allowed	Number of Messages Allowed	Print Control	Modifiable Entry	Print Buffer Content	Traceback Allowed	Standard Corrective Action	User Exit[7]
206	10	5	NA	Yes	NA	Yes	Yes	No
207	10	5	NA	Yes	NA	Yes	Yes	No
208	Unlimited	5	NA	Yes	NA	Yes	Yes	No
209	10	5	NA	Yes	NA	Yes	Yes[1]	No[1]
210	Unlimited	10	NA	Yes	NA	Yes	Yes[1]	No
211	10	5	NA	Yes	NA	Yes	Yes	No
212	10	5	No[2]	Yes	Yes	Yes	Yes	No
213	10	5	NA	Yes	NA	Yes	Yes	No
214	10	5	NA	Yes	NA	Yes	Yes	No
215	Unlimited	5	NA	Yes	Yes	Yes	Yes	No
216	10	5	NA	Yes	NA	Yes	Yes[3]	No
217	1[4]	1	NA	Yes	NA	Yes	Yes	No
218	10[5]	5	NA	Yes	Yes[5]	Yes	Yes	No
219	10[6]	5	NA	Yes	NA	Yes	Yes	No
220	10	5	NA	Yes	NA	Yes	Yes	No
221	10	5	NA	Yes	Yes	Yes	Yes	No
222	10	5	NA	Yes	Yes	Yes	Yes	No
223	10	5	NA	Yes	Yes	Yes	Yes	No
224	10	5	NA	Yes	Yes	Yes	Yes	No
225	10	5	NA	Yes	Yes	Yes	Yes	No
226	10	5	NA	Yes	NA	Yes	Yes	No
227	10	5	NA	Yes	NA	Yes	Yes	No
228	10	5	NA	Yes	NA	Yes	Yes	No
229	10	5	NA	Yes	NA	Yes	Yes	No
230	1	1	NA	No	NA	Yes	No	No
231	10	5	NA	Yes	NA	Yes	Yes	No
232	10	5	NA	Yes	NA	Yes	Yes	No
233-237	10	5	NA	Yes	NA	Yes	Yes	No
240	1	1	NA	No	NA	Yes	No	No
241-301	10	5	NA	Yes	NA	Yes	Yes	No

[1] No corrective action is taken except to return to execution. For boundary alignment, the corrective action is part of the support for misalignment.

[2] If a print control character is not supplied, the overflow line is not shifted to incorporate the print control character. Thus, if the device is tape, the data is intact, but if the device is a printer, the first character of the overflow line is not printed, but instead is treated as the print control. Unless the user has planned the overflow, the first character would be random and thus the overflow print line control can be any of the possible ones. It is suggested that when the device is a printer, the option be changed to single space supplied.

[3] Corrective action consists of return to execution for SLITE.

[4] It is not considered an error if the END parameter is present in a READ statement. No message or traceback is printed and the error count is not altered.

[5] For an I/O error, the buffer may have been partially filled or not filled at all when the error was detected. Thus, the buffer contents could be blank when printed. When an ERR parameter is specified in a READ statement, it is honored even though the error occurrence is greater than the amount allowed.

[6] The count field does not necessarily mean that up to 10 missing FILEDEF commands will be detected in a single debugging run, since a single WRITE performed in a loop could cause 10 occurrences of the message for the same missing FILEDEF command.

[7] The system generation process cannot create option table entries with user-exit address specified. A user exit must be specified at a later time.

Figure 28. Option Table Default Values

Features Available with the FORTRAN IV Library (MOD II) Only

The following features are supported by the H Extended compiler only.

Automatic Function Selection

The automatic function selection facility provides a concise set of generic names for built-in and library functions, which are used in place of the larger set of specific (data-type dependent) function names. Automatic function selection is requested by a new specification statement, GENERIC. The user's task of referring to built-in and library functions is now simplified because the same name can always be used for a function, even though the type of the function and the type of its arguments may vary with each use. Without automatic function selection, different names would have to be coded, depending upon the type of the function and its arguments.

Automatic Precision Increase Facility

The Automatic Precision Increase facility of the FORTRAN compiler automatically converts single precision floating point calculations to double precision and/or double precision to extended precision. It is designed to be used with programs originally written for earlier computers that offered greater precision than that available with System/360; the conversion facility may be used to convert programs where this extra precision may be of critical importance.

The facility is not meant to be used with new programs (those written for System/360 compilers). If such programs require operations with greater precision, they should be coded using the convert new programs, the cost in programmer and compilation time and the increase in storage space makes its use for this purpose inefficient.

No recoding of source programs is necessary to take advantage of the facility. Conversion is requested through the FORTHX command AUTODBL option at compilation time. The automatic precision increase facility should be considered as a tool for precision conversion; use of the facility does not assure that every FORTRAN source program, so converted, will execute correctly at the higher precision.

Precision Conversion Process

The conversion process comprises two functions: promotion and padding. Promotion is the process of converting items from one precision to a higher precision, for example, from single precision to double precision. The promotion function is described in greater detail below. Padding is the process of doubling the storage size of non-promoted items. Padding helps the user preserve the size relationships between promoted and non-promoted items sharing storage.

Promotion

The user may request either or both of the following conversions:

1. Single precision items to be promoted to double precision items, that is, REAL*4 to REAL*8 and COMPLEX*8 to COMPLEX*16.

2. Double precision items to be promoted to extended precision items, that is, REAL*8 to REAL*16 and COMPLEX*16 to COMPLEX*32.

Note that single precision items cannot be increased directly to extended precision items.

Constants, variables, and functions are promoted as follows:

Constants: Single-precision real and complex constants are promoted to double precision. Double-precision real and complex constants are promoted to extended precision. Logical and integer constants are not affected.

Examples of promoted constants are:

Constant	Promoted Form of Constant
3.0	3.0D0
4.24E5	4.24D5
4.24D5	4.24Q5
(3.2, 3.1416E0)	(3.2D0, 3.1416D0)

Variables: REAL*4 and COMPLEX*8 variables are promoted to REAL*8 and COMPLEX*16, respectively.

Examples of promoted variables are:

Variable	Promoted Form of Variable
REAL STAR, MOON, PLANET IMPLICIT REAL*8(S,T,U)	REAL*8 STAR, MOON, PLANET IMPLICIT REAL*16(S,T,U)
COMPLEX*8 A,B,C,D	COMPLEX*16 A,B,C,D

Functions: The correct FORTRAN-supplied functions are substituted when a program is converted. For example, a reference to SIN causes the DSIN function to be substituted if double precision calculation is to be preformed; a reference to DINT causes QINT to be substituted if extended precision calculation is performed. Figure 29 lists FORTRAN-supplied built-in functions that are substituted. Figure 30 lists FORTRAN-supplied library functions that are substituted. Function *values* are promoted in the same manner as constants; that is, single precision values are promoted to double precision, double precision values are promoted to extended precision.

Previously compiled subprograms must be recompiled to be converted to the correct precision. For example, if a user-supplied subprogram accepts only single precision arguments and is to be used with a program being converted to double precision, it must be recompiled using API to accept double precision arguments.

Effect of the AUTODBL and ALC Options on Automatic Precision Increase

The programmer requests automatic precision increase and padding through the AUTODBL and ALC options of the FORTHX command.

AUTODBL Option

The AUTODBL option to indicates the form that the conversion will take and the ALC subparameter to indicate whether storage alignment is to take place.

The AUTODBL option has the following format:

```
{AUTODBL|AD}        ( { NONE   } )
                      { DBLPAD }
                      { DBLPAD4}
                      { DBLPAD8}
                      { DBL    }
                      { DBL4   }
                      { DBL8   }
                      { abcde  }
```

Single Precision Function			Corresponding Double Precision Function			Corresponding Extended Precision Function		
Name	*Argument Type*	*Function Value Type*	*Name*	*Argument Type*	*Function Value Type*	*Name*	*Argument Type*	*Function Value Type*
AMOD	REAL*4	REAL*4	DMOD	REAL*8	REAL*8	QMOD	REAL*16	REAL*16
ABS	REAL*4	REAL*4	DABS	REAL*8	REAL*8	QABS	REAL*16	REAL*16
INT	REAL*4	INT*4	IDINT	REAL*8	INT*4	IQINT	REAL*16	INT*4
AINT	REAL*4	REAL*4	DINT	REAL*8	REAL*8	QINT	REAL*16	INT*
AMAXO[1]	INT*4	REAL*4						
AMAX1	REAL*4	REAL*4	DMAX1	REAL*8	REAL*8	QMAX1	REAL*16	REAL*16
MAX1[1]	REAL*4	INT*4						
AMINO[1]	INT*4	REAL*4						
AMIN1	REAL*4	REAL*4	DMIN1	REAL*8	REAL*8	QMIN1	REAL*16	REAL*16
MIN1[1]	REAL*4	INT*4						
FLOAT	INT*4	REAL*4	DFLOAT	INT*4	REAL*8	QFLOAT	INT*4	REAL*16
IFIX	REAL*4	INT*4	IDINT	REAL*8	INT*4	IQINT	REAL*16	INT*4
HFIX[1]	REAL*4	INT*2						
SIGN	REAL*4	REAL*4	DSIGN	REAL*8	REAL*8	QSIGN	REAL*16	REAL*16
DIM	REAL*4	REAL*4	DDIM	REAL*8	REAL*8	QDIM	REAL*16	REAL*16
REAL	COMPLEX*8	REAL*4	DREAL	COMPLEX*16	REAL*8	QREAL	COMPLEX*32	REAL*16
AIMAG	COMPLEX*8	REAL*4	DIMAG	COMPLEX*16	REAL*8	QIMAG	COMPLEX*32	REAL*16
CMPLX	REAL*4	COMPLEX*8	DCMPLX	REAL*8	COMPLEX*16	QCMPLX	REAL*16	COMPLEX*32
CONJG	COMPLEX*8	COMPLEX*8	DCONJG	COMPLEX*16	COMPLEX*16	QCONJG	COMPLEX*32	COMPLEX*32

[1] The corresponding double precision function does not exist by name, but the single precision function is expanded as though the double precision function existed.

Figure 29. Built-In Functions – Substitution of Single and Double Precision

Single Precision Function			Corresponding Double Precision Function			Corresponding Extended Precision Function		
Name	*Argument Type*	*Function Value Type*	*Name*	*Argument Type*	*Function Value Type*	*Name*	*Argument Type*	*Function Value Type*
EXP	REAL*4	REAL*4	DEXP	REAL*8	REAL*8	QEXP	REAL*16	REAL*16
CEXP	COMPLEX*8	COMPLEX*8	CDEXP	COMPLEX*16	COMPLEX*16	CQEXP	COMPLEX*32	COMPLEX*32
ALOG	REAL*4	REAL*4	DLOG	REAL*8	REAL*8	QLOG	REAL*16	REAL*16
CLOG	COMPLEX*8	COMPLEX*8	CDLOG	COMPLEX*16	COMPLEX*16	CQLOG	COMPLEX*32	COMPLEX*32
ALOG10	REAL*4	REAL*4	DLOG10	REAL*8	REAL*8	QLOG10	REAL*16	REAL*16
ARSIN	REAL*4	REAL*4	DARSIN	REAL*8	REAL*8	QARSIN	REAL*16	REAL*16
ARCOS	REAL*4	REAL*4	DARCOS	REAL*8	REAL*8	QARCOS	REAL*16	REAL*16
ATAN	REAL*4	REAL*4	DATAN	REAL*8	REAL*8	QATAN	REAL*16	REAL*16
ATAN2	REAL*4	REAL*4	DATAN2	REAL*8	REAL*8	QATAN2	REAL*16	REAL*16
SIN	REAL*4	REAL*4	DSIN	REAL*8	REAL*8	QSIN	REAL*16	REAL*16
CSIN	COMPLEX*8	COMPLEX*8	CDSIN	COMPLEX*16	COMPLEX*16	CQSIN	COMPLEX*32	COMPLEX*32
COS	REAL*4	REAL*4	DCOS	REAL*8	REAL*8	QCOS	REAL*16	REAL*16
CCOS	COMPLEX*8	COMPLEX*8	CDCOS	COMPLEX*16	COMPLEX*16	CQCOS	COMPLEX*32	COMPLEX*32
TAN	REAL*4	REAL*4	DTAN	REAL*8	REAL*8	QTAN	REAL*16	REAL*16
COTAN	REAL*4	REAL*4	DCOTAN	REAL*8	REAL*8	QCOTAN	REAL*16	REAL*16
SQRT	REAL*4	REAL*4	DSQRT	REAL*8	REAL*8	QSORT	REAL*16	REAL*16
CSQRT	COMPLEX*8	COMPLEX*8	CDSQRT	COMPLEX*16	COMPLEX*16	CQSQRT	COMPLEX*32	COMPLEX*32
TANH	REAL*4	REAL*4	DTANH	REAL*8	REAL*8	QTANH	REAL*16	REAL*16
SINH	REAL*4	REAL*4	DSINH	REAL*8	REAL*8	QSINH	REAL*16	REAL*16
COSH	REAL*4	REAL*4	DCOSH	REAL*8	REAL*8	QCOSH	REAL*16	REAL*16
ERF	REAL*4	REAL*4	DERF	REAL*8	REAL*8	QERF	REAL*16	REAL*16
ERFC	REAL*4	REAL*4	DERFC	REAL*8	REAL*8	QERFC	REAL*16	REAL*16
GAMMA[1]	REAL*4	REAL*4	DGAMMA[1]	REAL*8	REAL*8			
ALGAMA[1]	REAL*4	REAL*4	DLGAMA[1]	REAL*8	REAL*8			
CABS	COMPLEX*8	REAL*4	CDABS	COMPLEX*16	REAL*8	CQABS	COMPLEX*32	REAL*16

[1] The extended precision equivalences of these functions do not exist. In promoting REAL*8 to REAL*16, the double precision function will be used.

Figure 30. Library Functions – Substitution of Single and Double Precision

where:

NONE - indicates no conversion. This is the default condition.

DBLPAD - indicates promotion and padding of single and double precision items. REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 types are converted. Items of other types are padded if they share storage space with converted items.

DBLPAD4 - indicates promotion of single precision items only, and padding of other items that share storage with promoted items.

DBLPAD8 - indicates promotion of double precision items only, and padding of other items that share storage with promoted items.

Note: The promotion and padding options, DBLPAD, DBLPAD4, and DBLPAD8 ensure that the storage-sharing relationship that existed prior to conversion is maintained. Note, however, that padding reduces the efficiency of input/output operations for padded arrays.

DBL - indicates promotion (but no padding) of both single and double precision items. Items of REAL*4 and COMPLEX*8 types are converted to REAL*8 and COMPLEX*16 types are converted to REAL*16 and COMPLEX*32.

DBL4 - indicates promotion of single precision items only.

DBL8 - indicates promotion of double precision items only. If AUTODBL is specified, and an error in coding the parameter is detected, the compiler substitutes the option DBLPAD8 as a default.

Note: For most programs, one of the above forms is sufficient. The following form offers greater flexibility to the user who wishes to tailor the conversion process to a particular program; however, it also increases the chance of error and should be used with care.

abcde - indicates that the program is to be converted according to the value of abcde, a five-position field. Each posision is coded with a numeric value that specifies how a particular conversion function is to be performed.

The leftmost position *(a)* describes the promotion function, that is, whether promotion is to occur and, if so, which items are to be promoted. The second position *(b)* describes the padding function, that is, whether padding is to occur and, if so, the sections in the program (such as COMMON or argument lists) where padding is to take place. The third, fourth, and fifth positions describe whether padding is to occur for particular types (LOGICAL, INTEGER, and REAL, respectively) within the program sections specified in position *b*.

All five positions must be coded; if a function is to be omitted, the corresponding position is coded with a zero. The values for each position are as follows:

- Position *a*, the promotion function:

Value	Meaning
0	No promotion
1	Promote REAL*4 and COMPLEX*8 items only
2	Promote REAL*8 and COMPLEX*16 items only
3	Promote all real and complex items

- Position *b*, the padding function:

Value	Meaning
0	No padding
1	Pad COMMON statement and argument list variables
2	Pad EQUIVALENCE statement variables equivalenced to promoted variables
3	Pad COMMON and EQUIVALENCE statement variables and argument list variables
4	Pad EQUIVALENCE statement variables that do not relate to variables in COMMON statements
5	Pad all variables

The code specified in this position determines in which areas of a program the padding requested by positions *c* to *e* is to take place.

- Position *c*, padding logical variables in program sections specified in position *b:*

Value	Meaning
0	Pad no logical variables
1	Pad LOGICAL*1 variables only
2	Pad LOGICAL*4 variables only
3	Pad all logical variables

- Position *d*, padding integer variables in program sections specified in position *b:*

Value	Meaning
0	Pad INTEGER*2 variables only

2	Pad INTEGER*4 variables only
3	Pad all integer variables.

- Position *e,* padding real and complex variables in program sections specified in position *b:*

Value	Meaning
0	Pad no real or complex variables
1	Pad REAL*4 and COMPLEX*8 variables
2	Pad REAL*8 and COMPLEX*16 variables
3	Pad REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16 variables
4	Pad REAL*16 and COMPLEX*32 variables
5	Pad REAL*4, COMPLEX*8, REAL*16, and COMPLEX*32 variables
6	Pad REAL*8, REAL*16, COMPLEX*16, and COMPLEX*32 variables
7	Pad all real and complex variables

Note that promotion overrides padding. For example, if the first position specifies promotion to occur for single precision items, REAL*4 and COMPLEX*8 items are promoted regardless of the padding function specified in position *e.*

Examples:

The AUTODBL *(abcde)* settings that correspond to the mnemonic options are:

abcde Setting	Correspond to the Mnemonic
(00000)	NONE
(30000)	DBL
(10000)	DBL4
(20000)	DBL8
(33334)	DBLPAD
(13336)	DBLPAD4
(23335)	DBLPAD4

The following examples illustrate other possible combinations of the AUTODBL *(abcde)* format.

Example 1:

```
AUTODBL(12330)
```

Promotion is performed and padding is performed for all EQUIVALENCE statements, logical variables, and integer variables.

Example 2:

```
AUTODBL(01001)
```

No promotion is performed, but padding is performed for all REAL*4 and COMPLEX*8 variables in common blocks and argument lists. This code setting permits a program not requiring double precision accurracy to link with a subprogram compiled with the option AUTODBL(DBL).

Example 3:

AUTODBL(01337)

No promotion is performed, but padding is performed for all integer, logical, real, and complex variables that are in COMMON or are used as subprogram arguments. This code setting permits a non-converted program to link with a program converted with the option AUTODBL(DBLPAD4).

ALC Option

The ALC option is used to specify storage alignment. It has the following format:

{ ALC
<u>NOALC</u> }

where:

ALC - indicates that storage alignment is to take place.

NOALC - indicates that storage alignment is not to take place. This is the default.

Ordinarily, to increase execution-time efficiency, COMMON statements are coded so that variables in COMMON blocks are aligned on proper boundaries: doubleword variables on doubleword boundaries, fullword variables on fullword boundaries, and halfword variables on halfword boundaries. When the conversion facility is used, these alignments may become altered. The ALC option restores alignment.

The ALC option should be used with care for it may cause previously mathced COMMON blocks to become mismatched. Consider the two COMMON statements below where the variable INTER is to be shared.

Program 1	**Program 2**
REAL*8 R8	REAL*4 R4
COMMON/X/A, R8, INTER	COMMON/X/Z,I,R4, INTER

With neither the AUTODBL nor the ALC option specified, both occurrences of the variable INTER will be at an offset of 12 bytes from the start of COMMON block X.

If ALC alone is used, INTER would be 16 bytes from the start of COMMON X in Program 1 since R8 would have been placed on a double word boundary. COMMON X in Program 2 would have been unaffected.

If AUTODBL(DBL4) and ALC are specified, INTER would be 16 bytes from start of block X in Program 1 and 24 bytes from start in Program 2. (This is because of the promotion of REAL*4 to REAL*8 and subsequent alignment.)

It is recommended that ALC be used only when the COMMON variables are identical in type.

Programming Considerations with API

This section provides a brief discussion of how use of the Automatic Precision Increase facility affects program processing.

Effect on COMMON or EQUIVALENCE Data Values

Promotion and padding operations preserve the storage sharing relationships that existed before conversion. However, in items that share storage data values are preserved only for the following:

1 Variables having the same length

2 Real and complex variables having the same precision

The following items retain value sharing relationships:

LOGICAL*4 and INTEGER*4 (same length)

REAL*4 and COMPLEX*8 (same precision)

The following items do not retain value sharing relationships:

INTEGER*2 and INTEGER*4 (different lengths)
REAL*8 and COMPLEX*8 (different precisions)

Effect on Literal Constants

Care should be exercised when specifying literal constants as data initialization values for promoted or padded variables, as subprogram arguments, or in NAMELIST input. For example, literals should be entered into arrays on an element by element basis rather than as one continuous string.

Example:

```
DIMENSION A(2), B(2)
DATAA/'ABCDEFGH'/,B(1)/'IJKL'/,B(2)/'MNOP'/
```

Array B will be initialized correctly but not array A, because padding takes place at the end of each element.

Effect on Programs Calling Subprograms

FORTRAN main programs and subprograms must be converted so that variables in COMMON retain the same relationship to guarantee correct linkage during execution. The recommended procedure is to compile all program units using AUTODBL (DBLPAD). If an option other than DBLPAD is selected, care must be taken if the COMMON variables in one program unit differ from those in another; COMMON variables that are not to be promoted should be padded.

Any non-FORTRAN external subprogram called by a converted program unit should be recoded to accept padded and promoted arguments.

Effect on FORTRAN Library Subprograms

1 If a call to a FORTRAN library subprogram contains promoted arguments, the next higher precision subprograms are substituted for the original ones. The external symbol dictionary, used by the CMS loader to resolve references between program units, will contain the double and extended precision names for each single and double precision library program promoted.

2 If you have supplied your own function for a FORTRAN-supplied function, but has neglected to detach the name through an EXTERNAL statement, the wrong function may be executed.

Example: AUTODBL(DBL4)

```
      REAL*4 X,Y
    4 Y=SIN(X)
      STOP
      END
       .
       .
       .
      FUNCTION SIN(X)
       .
       .
       .
      RETURN
      END
```

In this example, because the compiler cannot recognize SIN as a user-supplied function, it substitutes the name of the FORTRAN-supplied function DSIN in the statement labeled 4. However, the compiler does not change the function definition statement; the name remains SIN. At execution time the

user-supplied function SIN is ignored and the FORTRAN-supplied function DSIN is executed in its place.

The programmer can avoid this confusion either by making sure he detaches the name SIN, preceded by an ampersand, in an EXTERNAL statement or by changing the name of the function to DSIN.

Effect on CALL DUMP or CALL PDUMP Statements

If a CALL DUMP or CALL PDUMP statement specifies a dump format of either REAL*4 or COMPLEX*8, output from a promoted or padded program is displayed as two single precision numbers rather than as one double precision number.

For variables that are promoted, the first number is *approximately* the value of the stored variable; the second number is meaningless.

The variables that are padded, the first number is *exactly* the value of the stored variable; the second number is meaningless.

Effect on Direct-Access Input/Output Processing

When a DEFINE FILE statement has been specified, any record exceeding the maximum specified record length causes record overflow to occur. For converted programs, the programmer should check the record size coded in the statement to determine if it can handle the increased record lengths. If not sufficient, the size should be increased appropriately.

Effect on Unformatted Input/Output Data Sets

Unformatted input/output data sets that have not been converted are not acceptable to converted programs if the I/O list contains promoted variables.

Effect on the Storage Map

The storage map produced by the MAP option of the FORTHX command contains the following codes:

Code	Meaning
D	Promoted variable
P	Padded variable
*	Promoted library function name

Extended Precision

Through its extended-precision cabability, the compiler can recognize and process two additional data types in the FORTRAN source language: REAL*16 and COMPLEX*32. As a result, programs that were heretofore limited by insufficient precision can now be run using these data types. For real and complex data items, the maximum number of storage locations that can be allocated per data item is twice the previous maximum. Also, the FORTRAN-supplied functions required to support the extended-precision data types are provided, with two more exceptions. Extended-precision equivalents of the GAMMA and ALGAMA functions are not included.

External Statement Extension

An extension to the EXTERNAL statement enables the user to "detach" the names of FORTRAN library subprograms. Detachment of a subprogram name causes that name to be dissociated with the FORTRAN-supplied library subprogram of the same name; instead, it is considered to be the name of user-supplied subprogram. This extension is provided when you prefix the special character & to a subprogram name when it appears in the EXTERNAL statement. The extension enables you to supply your own subprograms in place of identically named FORTRAN library subprograms, with the assurance that the compiler will interpret all subprogram references correctly.

FORTRAN IV(G1) Compiler

The FORTGI command invokes the IBM FORTRAN IV (G1) compiler, which will compile the FORTRAN source program contained in the CMS file that you identify in the command. The FORTGI command allows you to specify a set of options governing compiler operation and output; however, should you omit one or all of the options, defaults are assumed for you. If you include options that are not valid for the FORTRAN IV (G1) compiler or if you misspell any options, a diagnostic message is typed out at your terminal (see Appendix F for more information). In the following illustrations and descriptions all defaults for the compiler default options are underlined.

FORTGI Command Format[1]

Figure 31 shows the format of the FORTGI command the options that are available.

```
FORTGI filename     ( [BCD|EBCDIC] [DECK|NODECK] [ID|NOID] [LINECNT(nn|50)]
                    [LIST|NOLIST] [LOAD|NOLOAD] [MAP|NOMAP] [NAME(name|MAIN)]
                    [DISK|PRINT|NOPRINT] [SOURCE|NOSOURCE] [TERM|NOTERM]
                    [TEST|NOTEST] )
```

Figure 31. Format of the FORTGI Command for the FORTRAN IV (G1) Compiler

- Identifying the Compiler to be Used

  FORTGI -- The word FORTGI identifies the command and must be typed as shown.

- Specifying a File for Compilation

  filename -- Specified the name of the file to be compiled. The file specified must have a filetype of FORTRAN or it will not be recognized as input for the FORTRAN IV (G1) compiler.

Note: You must insure that the file named does not contain any statements that are not acceptable to the FORTRAN IV (G1) compiler (for example, GENERIC statements).

- Character Code of the Source Program

  BCD -- The source program to be compiled is written in BCD.

---

[1]The material in this section may be reproduced for internal use; it may not be offered for resale.

Note: The CMS COPYFILE command with the EBCDIC option can be used to convert a file containing BCD code to a file in EBCDIC, thus eliminating the need for this option.

EBCDIC -- The source program to be compiled is written in EBCDIC.

If you omit this option, the compiler will assume that your source program is written in EBCDIC.

- Producing a Card Deck for Your TEXT File

  DECK -- The executable code produced by the compiler will be punched into a card deck in your computing center.

  NODECK -- The executable code produced by the compiler will not be punched into a card deck.

  If you omit this option, the compiler assumes NODECK.

- Producing a Listing File for Your Program

  DISK -- The compiler will place a copy of your LISTING file on a disk.

  PRINT -- The compiler will print your LISTING file on an offline printer.

  NOPRINT -- No LISTING file will be produced.

  If you omit this option, the compiler assumes DISK.

- Internal Statement Numbers (ISN)

  ID -- The compiler will generate internal statement numbers for statements that call subroutines or contain external function references. The ID option allows the translate function of the Mod I library to display the internal statement numbers of the linkages that are in effect at the time of an error.

  NOID -- The compiler will not generate internal statement numbers.

  If you omit this option, the compiler assumes NOID.

- Number of Lines to be Printed on Each Listing Page

  LINECNT *nn* -- The source listing for your program is to be printed with a maximum of *nn* lines per page. You may specify any number for *nn* from 1 to 99.

  If you omit this option, the compiler assumes 50 lines per page.

- Producing a Listing of Your Object Module

  LIST -- The compiler will include, in the LISTING file, a pseudo-assembler listing of the translated statements contained in the TEXT file.

NOLIST -- The pseudo-assembler listing for your program will not be included in the LISTING file.

If you omit this option, the compiler assumes NOLIST.

- Producing Executable Object Code for Your Program

  LOAD -- The compiler will create a TEXT file which contains the executable code of your FORTRAN source program.

  NOLOAD -- No TEXT file is produced.

  If you omit this option, the compiler assumes LOAD.

- Producing a Storage Map for Your Source Program

  MAP -- The compiler will generate tables showing the name and location in your program of any array, COMMON, EQUIVALENCE, and scalar variables and FORMAT, NAMELIST, and subprogram statements. These tables will be included in your listing.

  NOMAP -- The compiler will not create the storage tables for you.

  If you omit this option, the compiler will assume NOMAP.

- Naming Your Program

  NAME *name* -- The name represented by *name* will be assigned by the compiler to the executable code it produces. You may specify from 1 to 6 characters for *name.*

  If you omit this option and the NOTEST option is in effect, the compiler assigns the filename as the name of your executable code. If the TEST option has been specified, MAIN is assumed.

- Producing a Source Program Listing

  SOURCE -- The compiler will include a copy of your FORTRAN source program in the LISTING file that it produces for you.

  NOSOURCE -- A copy of your source program will not be included in the LISTING file.

  If you omit this option, the compiler assumes SOURCE.

- Typing Compiler Error Messages at Your Terminal

  TERM -- Any erroneous statements detected in your FORTRAN program and the corresponding messages will be typed at your terminal.

  NOTERM -- Errors and messages will not be typed at your terminal, but will appear in your listing as usual.

  If you omit this option, the compiler assumes TERM.

- Making Your Programs Acceptable for Use with FORTRAN Interactive Debug

  TEST -- The object code produced for your program will contain additional linkages to make it acceptable to execute under FORTRAN Interactive Debug. When this option is specified, the LOAD option is assumed. See the publication *IBM FORTRAN Interactive Debug for OS(TSO) and VM/370 (CMS) Terminal User's Guide,* Order No. SC28-6885 for information on using FORTRAN Interactive Debug.

  NOTEST -- The object code does not include additional linkages for FORTRAN Interactive Debug.

  If you omit this option, the compiler assumes NOTEST.

Output from the FORTRAN IV (G1) Compiler

The FORTRAN IV (G1) compiler may produce a LISTING file that contains any errors detected during compilation, and informative and diagnostic messages. It may include a copy of your source statements, a storage map of the variables that you used in your program, and a pseudo-assembler listing of the code that was produced for your program by the compiler. In addition, you can direct that any error messages included in the LISTING FILE by typed at your terminal. You can use CMS commands to print the LISTING file either on a printer or at your terminal. A second file, the TEXT file, may also be produced that will contain the actual executable code. The FORTRAN IV (G1) compiler can produce a printed listing or a punched card form of your TEXT file. See Figure 32 for a summary of the FORTRAN IV (G1) compiler options and their effect on output.

Option	LISTING File	TEXT File	Terminal Response
MAP	Includes address tables of FORTRAN variables, and NAMELIST and FORMAT statements		
DECK		Punches a copy of this file offline	
LIST	Includes a pseudo-assembler listing of the TEXT file		
SOURCE	Includes the source code from the CMS FORTRAN source file		
PRINT	Creates this file and prints a copy offline		
DISK	Creates this file and writes a copy on an available disk		
TERM			Prints error messages at the terminal
LOAD		Creates this file	

Figure 32. The Effect of Various Compiler Options on Compiler Output (G1)

FORTRAN IV (G1) Listing File

A LISTING file is produced by the FORTRAN IV (G1) compiler unless the NOPRINT option is specified. It contains the following:

- Informative messages that indicate the status of the compilation.
- Any errors detected during compilation and the corresponding diagnostic messages. For a detailed description of the diagnostic messages produced by the FORTRAN IV (G1) compiler, refer to the publication *IBM FORTRAN IV (G1) Processor and TSO FORTRAN Prompter for OS and VM/370 (CMS) Installation Reference Material,* Order No. SC28-6856.
- Optional output as determined by the options you can specify with the FORTGI command or their defaults.

Informative Messages

The informative messages included in your listing identify the compiler used, the name of your program, the Julian date, and the time of day (based on a 24-hour clock) that the compilation was begun. A list of the options in effect and the compiler statistics are also provided. See the part of Figure 33, labeled A for an illustration of the compiler informative messages.

Error Messages

Error messages produced by the FORTRAN IV (G1) compiler have two formats, depending on when the error was detected. Statements in which an error, such as syntax, is detected as the statement is being processed, are followed by a line that contains a $ sign positioned beneath each point at which an error is detected. This pointer line is followed by a line containing the number of the error (IGI *xxxI*) and the text of the diagnostic message. When more than one error occurs in a single course statement, the error messages following it are numbered consecutively from left to right.

Example:

```
                              .
                              .
                              .
0009          106     IF(L*K-I)1,2,4
0010                  CONTINUE
                $
******** 01)  IGI002I LABEL
                              .
                              .
                              .
```

If an error, such as an undefined label, is not detected until all the statements have been processed, the error message follows the source program and includes a list of any unresolved items.

Example:

```
                        .
                        .
                        .
IGI022I   UNDEFINED LABEL
          4
```

See the part of Figure 33, labeled B for the format of the error messages produced by the compiler. In addition to the diagnostic messages produced by the compiler, the CMS ready message indicates the highest severity level detected.

Printing Error Messages at Your Terminal

Since it is helpful in correcting your source program to get a copy of any errors and messages at your terminal as they are produced, the TERM default option types them out automatically for you. You may suppress these messages by specifying the NOTERM option with the FORTGI command.

Optional Output

Additional information can be included in your LISTING file depending on the defaults in effect or the options you specify with the FORTGI command. You can include the following:

- A list of your source statements
- A storage map.
- A pseudo-assembler listing of the executable code produced for your program.

Obtaining a Copy of Your Source Statements

Since the SOURCE option is the default for the FORTGI command, a copy of your source statements is included in the LISTING file automatically. See the part of Figure 33 labeled C for the format of the source listing. If you do not want your source statements included, you must specify the NOSOURCE option.

Obtaining a Storage Map

The storage map is a table that contains entries generated by the compiler for each of seven classifications of variables that you may have used in your program. The classifications are:

- Array Variables
- COMMON Variables
- EQUIVALENCE Variables
- FORMAT Statements
- NAMELIST Statements
- Scalar Variables
- Subprogram Statements

See the part of Figure 33 labeled D for the format of the map. It lists by classification each variable and its internal location. If you want the storage map included in the LISTING file, you must specify the MAP option with the FORTGI command.

Obtaining a Pseudo-assembler Listing of Your Executable Code

The pseudo-assembler listing contains your FORTRAN source statements after they have been translated into an executable form by the FORTRAN IV (G1) compiler. This listing represents the executable code in an assembler language format. It indicates the relative locations (in hexadecimal format), the compiler generated sequence numbers, the assembler language codes showing labels, op-codes and operands, and the BCD operands, which identifies any significant items (variables, entry points, or labels) referred to by the instruction. See the part of Figure 33 labeled E for the format of the object code listing. If you want a copy of your object code included in the LISTING file, you must specify the LIST option with the FORTGI command.

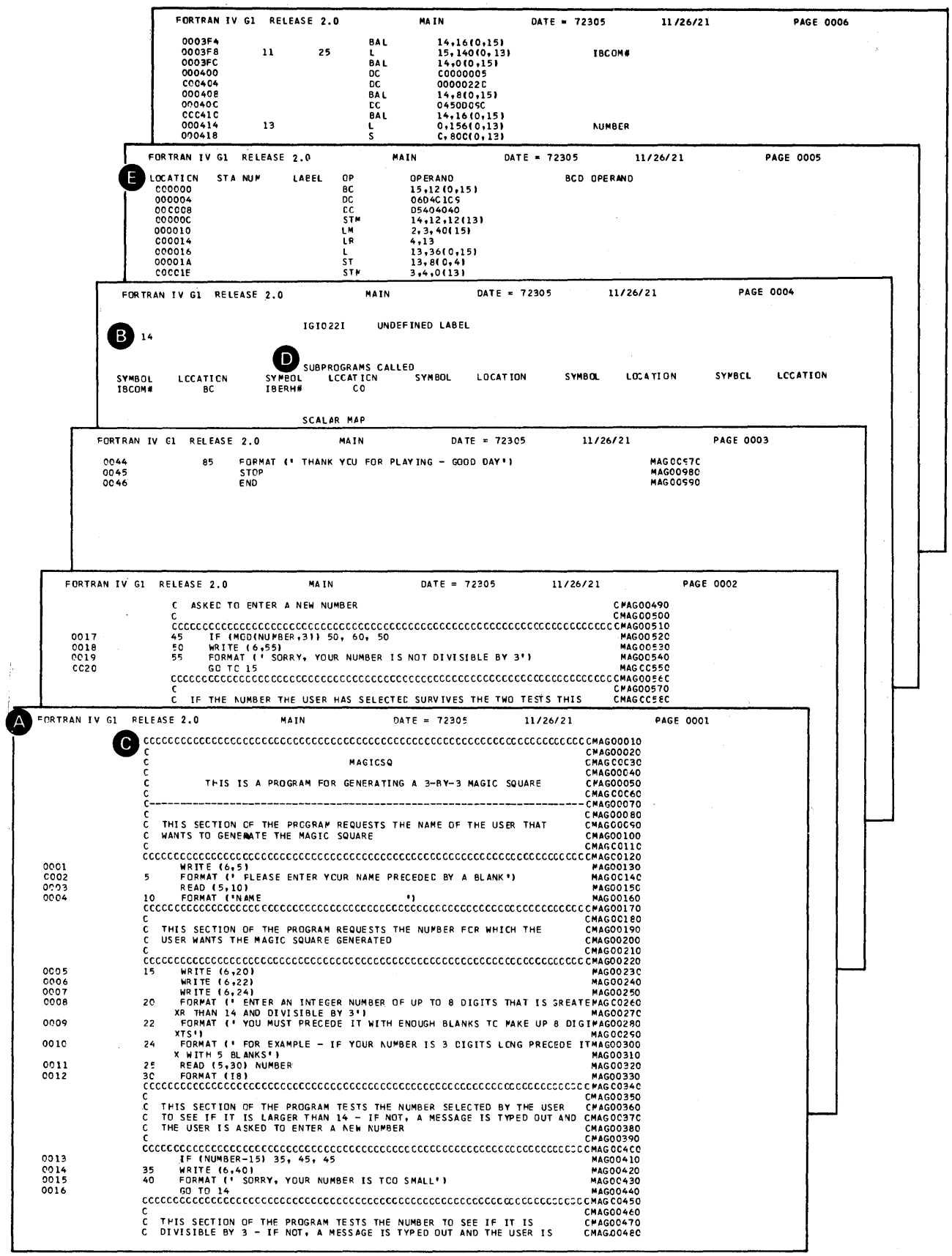
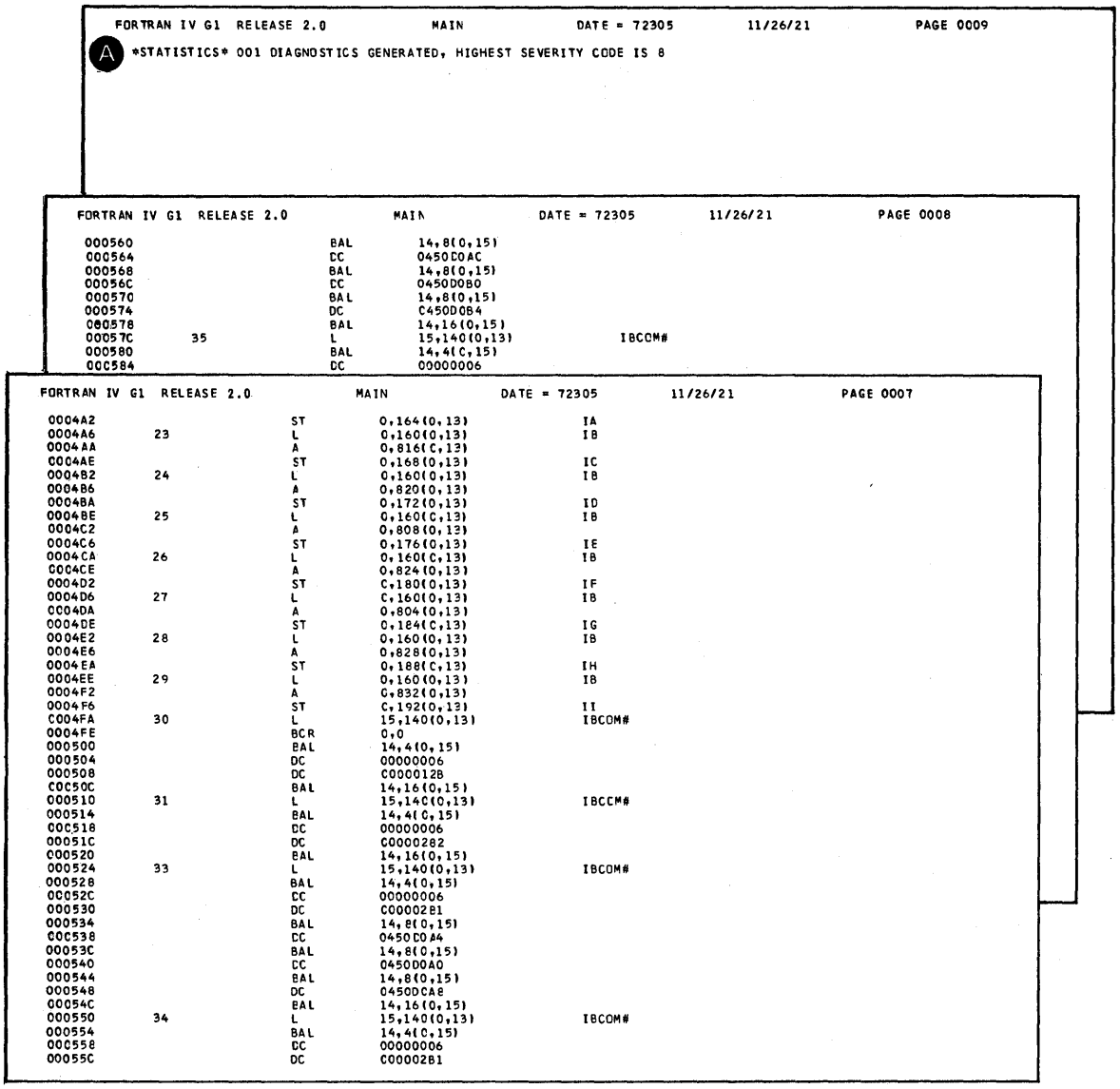


Figure 33. FORTRAN IV (G1) LISTING File (Part 1 of 2)



```
FORTRAN IV G1  RELEASE 2.0            MAIN            DATE = 72305        11/26/21          PAGE 0009

 A  *STATISTICS* 001 DIAGNOSTICS GENERATED, HIGHEST SEVERITY CODE IS 8
```

```
FORTRAN IV G1  RELEASE 2.0            MAIN            DATE = 72305        11/26/21          PAGE 0008

 000560                        BAL       14,8(0,15)
 000564                        DC        0450C0AC
 000568                        BAL       14,8(0,15)
 00056C                        DC        0450D0B0
 000570                        BAL       14,8(0,15)
 000574                        DC        C450D0B4
 000578                        BAL       14,16(0,15)
 00057C       35               L         15,140(0,13)          IBCOM#
 000580                        BAL       14,4(C,15)
 000584                        DC        00000006
```

```
FORTRAN IV G1  RELEASE 2.0            MAIN            DATE = 72305        11/26/21          PAGE 0007

 0004A2                        ST        0,164(0,13)           IA
 0004A6       23               L         0,160(0,13)           IB
 0004AA                        A         0,816(C,12)
 0004AE                        ST        0,168(0,13)           IC
 0004B2       24               L         0,160(0,13)           IB
 0004B6                        A         0,820(0,13)
 0004BA                        ST        0,172(0,13)           ID
 0004BE       25               L         0,160(C,13)           IB
 0004C2                        A         0,808(0,13)
 0004C6                        ST        0,176(0,13)           IE
 0004CA       26               L         0,160(C,13)           IB
 0004CE                        A         0,824(0,13)
 0004D2                        ST        C,180(0,13)           IF
 0004D6       27               L         C,160(0,13)           IB
 0004DA                        A         0,804(0,13)
 0004DE                        ST        0,184(C,13)           IG
 0004E2       28               L         0,160(0,13)           IB
 0004E6                        A         0,828(0,13)
 0004EA                        ST        0,188(C,13)           IH
 0004EE       29               L         0,160(0,13)           IB
 0004F2                        A         C,832(0,13)
 0004F6                        ST        C,192(0,13)           II
 0004FA       30               L         15,140(0,13)          IBCOM#
 0004FE                        BCR       0,0
 000500                        BAL       14,4(0,15)
 000504                        DC        00000006
 000508                        DC        C000012B
 00C50C                        BAL       14,16(0,15)
 000510       31               L         15,140(0,13)          IBCOM#
 000514                        BAL       14,4(C,15)
 00C518                        DC        00000006
 00051C                        DC        C0000282
 000520                        BAL       14,16(0,15)
 000524       33               L         15,140(0,13)          IBCOM#
 000528                        BAL       14,4(0,15)
 00052C                        DC        00000006
 000530                        DC        C00002B1
 000534                        BAL       14,8(0,15)
 00C538                        DC        0450C0A4
 00053C                        BAL       14,8(0,15)
 000540                        DC        0450D0A0
 000544                        BAL       14,8(0,15)
 000548                        DC        0450DCA8
 00054C                        BAL       14,16(0,15)
 000550       34               L         15,140(0,13)          IBCOM#
 000554                        BAL       14,4(C,15)
 00055B                        DC        00000006
 00055C                        DC        C00002B1
```

Figure 33. FORTRAN IV (G1) LISTING File (Part 2 of 2)

FORTRAN IV (G1) TEXT File

A TEXT file is created by the FORTRAN IV (G1) compiler whenever the LOAD default option is in effect. The file contains the compiled FORTRAN IV program that is identical to the object program produced by the FORTRAN IV (G1) compiler under OS.

The TEXT file contains the version of your program that can be executed by CMS or link edited by OS. You can issue a RUN command, a LOAD and a START command, or an EXEC command that specifies a file containing these commands and your program will begin executing. (Any additional FILEDEF commands required for the execution of the program must be issued before you can enter a START command.)

Obtaining a Punch Card Deck of Your Object Program

To have your TEXT file punched into a card deck, you must specify the DECK option with the FORTGI command. An alternate method is to use the PUNCH command specifying the filename of your file and a filetype of TEXT. This will punch the TEXT file that was created and placed on a disk by the LOAD option when specified during a compilation.

Example:

```
punch newprog text
```

Compiler Language Restrictions for FORTRAN IV (G1)

The following limitations are placed upon the FORTRAN IV language by the FORTRAN IV (G1) compiler:

- The maximum level of nesting for DO loops and implied DO statements is 25.
- The maximum level of nested references in an arithmetic statement function definition to other statement function subprograms is 25.
- The maximum number of expressions that can be nested is 100.
- The maximum number of continuation cards for a single statement is 19.
- The repetition field (a) for format codes in a FORMAT statement, if present, must be an unsigned integer constant less than 256.
- In literal constants in the source program, any valid card code is permissible, except a 12-11-0-7-8-9 punch (hexadecimal 'FF' or binary (integer) minus one, (-1)).

Refer to the publication *IBM System/360 and System/370 FORTRAN IV Language,* Order No. GC28-6515 for a complete description of the statements involved.

Code and Go FORTRAN IV Compiler

The GOFORT command invokes the IBM Code and Go FORTRAN IV compiler, which will compile the FORTRAN source program contained in any CMS files that you identify in the command. The Code and Go Compiler will accept free-form FORTRAN source code with 80-character lines and it allows you to execute your programs immediately after compilation without the necessity of issuing CMS commands to load and execute. You may include a set of options in the GOFORT command that govern compiler operation and output; however, should you omit one or all of the options, defaults are assumed for you. If you include options that are not valid for the Code and Go FORTRAN compiler or if you misspell any options, a diagnostic message is typed out at your terminal (see Appendix F for more information). In the following illustrations and descriptions all defaults for the compiler options are underlined.

GOFORT Command Format[1]

Figure 34 shows the format of the GOFORT command and the options that are available.

```
GOFORT filename    ( [BCD|EBCDIC] [DECK|NODECK] [FIXED|FREE] [GO|NOGO]
                   [LINECNT(nn|50)] [LMSG|SMSG] [DISK|PRINT|NOPRINT]
                   [SOURCE|NOSOURCE] [TEST|NOTEST] )
```

Figure 34. Format of the GOFORT Command for the Code and Go FORTRAN IV Processor

- Identifying the Compiler to be Used

  GOFORT -- The word GOFORT identifies the Code and Go FORTRAN IV compiler and must be typed as shown.

- Specifying a File for Compilation

  filename -- Specifies the name of the file to be compiled. The file specified must have a filetype of FORTRAN or FREEFORT or it will not be recognized as input for the Code and Go FORTRAN IV compiler. If the file has a filetype of FREEFORT, the FREE option must be specified.

  Note: You must insure that the file named does not contain any statements that are not acceptable to the Code and Go compiler (for example, GENERIC statements).

[1] The material in this section may be reproduced for internal use; it may not be offered for resale.

- Character Code of the Source Program

  BCD -- The source program to be compiled is written in BCD.

  Note: The CMS COPYFILE command with the EBCDIC option can be used to convert a file containing BCD code to a file in EBCDIC, thus eliminating the need for this option.

  EBCDIC -- The source program to be compiled is written in EBCDIC.

  If you omit this option, the compiler will assume that your source program is written in EBCDIC.

- Producing a TEXT File for your program

  DECK -- A TEXT file containing the executable code for your program is to be produced by the compiler

  NODECK -- A TEXT file will not be produced for your program

  If you omit this option, the compiler assumes NODECK. When the TEST option is in effect, DECK is assumed.

- Producing a Listing for Your Program

  DISK -- The compiler will place a copy of your LISTING file on a disk.

  PRINT -- The compiler will print your LISTING file on an offline printer.

  NOPRINT -- No LISTING file will be produced.

  If you omit this option, the compiler assumes DISK.

- Executing a TEXT file Immediately after Compilation

  GO -- The compiler will generate a TEXT file that is to be executed immediately after compilation without issuing additional CMS commands to load or execute it.

  NOGO -- The TEXT file produced will not be executed automatically. Additional CMS commands will be required to load and execute it.

  If you omit this option, the compiler assumes GO. When the TEST option is in effect, NOGO is assumed.

- Fixed of Free Form of the FORTRAN Source Code

  FIXED -- The source code is written in fixed-form format.

  FREE -- The source code is written in free-form format.

  If you omit this option, the compiler assumes that your source code is written in fixed-form format.

- Number of Lines to be Printed on Each Listing Page

  LINECNT *nn* -- The source listing for your program is to be printed with a maximum of *nn* lines per page. You may specify any number for *nn* from 1 to 99.

  If you omit this option, the compiler assumes 50 lines per page.

- Format of Compiler Error Messages

  LMSG -- The compiler will print error messages in a long and detailed form.

  SMSG -- The compiler will print error messages in a short and concise form.

  If you omit this option, the compiler assumes SMSG.

- Producing a Source Program Listing

  SOURCE -- The compiler will include a copy of your FORTRAN source program in the listing that it produces for you.

  NOSOURCE -- A copy of your source program will not be included in the listing.

  If you omit this option, the compiler will assume SOURCE.

- Making Your Programs Acceptable for Use with FORTRAN Interactive Debug

  TEST -- The object code produced for your program will contain additional linkages to make it acceptable to run under FORTRAN Interactive Debug. When this option is specified, the DECK and NOGO option is assumed. See the publication *IBM FORTRAN Interactive Debug for OS(TSO) and VM/370 (CMS)*, Order No. SC28-6885 for information on using FORTRAN Interactive Debug.

  NOTEST -- The object code does not include additional linkages for FORTRAN Interactive Debug.

  If you omit this option, the compiler assumes NOTEST.

Output from the Code and Go FORTRAN IV Compiler

The Code and Go FORTRAN IV compiler produces a LISTING file whenever errors are detected during compilation. It contains diagnostic messages. It may optionally include a copy of your source statements. A second file, the TEXT file, may be produced. The Code and Go compiler can produce a punched deck of cards of your TEXT file. See Figure 35 for a summary of the Code and Go FORTRAN IV compiler options and their effect on output.

Option	LISTING File	TEXT File	Terminal Response
SOURCE	Creates this file and Includes the source code from the CMS FORTRAN source file		
DECK		Creates this file	
PRINT	Creates this file and prints a copy offline		
DISK	Creates this file and writes a copy on an available disk. This file is created whenever errors are detected.		All error and diagnostic messages are always typed at the terminal

Figure 35. The Effect of Various Compiler Options on Compiler Output (Code & Go)

Code and Go FORTRAN IV LISTING File

A LISTING file is produced by the Code and Go FORTRAN IV compiler unless the NOPRINT option is in effect. It contains the following:

- Informative messages.
- Any errors detected during compilation and the corresponding diagnostic messages. For a detailed description of the diagnostic messages produced by the Code and Go FORTRAN IV compiler, refer to publication *IBM Code and Go FORTRAN IV Processor for OS and VM/370 (CMS) Installation Reference Material,* Order No. SC28-6859.
- Optional output as determined by the options you can specify with the GOFORT command or their defaults.

Informative Messages

The informative messages included in your listing identify the compiler used, the Julian date, and the time of day (based on a 24-hour clock) that the compilation was begun. See the part of Figure 36 labeled A for an illustration of the compiler informative message.

Error Messages

Error messages produced by the Code and Go FORTRAN IV compiler are listed in a group. They contain the error number (IGK*xxx* I) and the text of the message. When a message refers to a specific source statement, an internal sequence identification number is included to identify the statement in error. This number is printed after the message identification number and before the message text. The number is a system-assigned line number. All compiler messages include this number except the following: IGK192I through IGK200I, IGK405I, IGK412I through IGK418I, IGK475I through

IGK477I, and IBK586I. These messages refer to general conditions affecting compilation and not to specific source statements; therefore, sequence numbers are not relevant in these cases.

The text of the messages has two forms, long and short. The short form is automatically provided, since SMSG is the default for the GOFORT command. You can request a longer, more detailed form of the error message by specifying the LMSG option.

Examples:

Short form of the message

```
IGK420I 00000123 NO STMT NMBR
```

Long form of the message

```
IGK420I 00000123 STMT FOLLOWING A TRANSFER OF CONTROL
        HAS NO STMT NMBR
```

See the part of Figure 36 labeled B for an illustration of the error messages.

Optional Output

You may include in the LISTING file a copy of your source program.

Obtaining a Copy of Your Source Statements

Since the SOURCE option is the default for the GOFORT command, you will have a copy of your source statements included in the LISTING file automatically. See the part of Figure 36 labeled C for the format of the source listing. If you do not want your source statements included, you must specify the NOSOURCE option.

Code and Go FORTRAN IV TEXT File

A TEXT file is created by the Code and Go FORTRAN IV compiler, whenever the DECK option is in effect. The file contains the compiled FORTRAN IV program that is identical to the object program produced by the Code and Go compiler under OS.

The text file contains the version of your program that can be executed by CMS. When the NOGO option of the GOFORT command is in effect, you can issue a RUN command, a LOAD and a START command or an EXEC command that specifies a file containing these commands, and your program will begin executing. (Any additional FILEDEF commands required for the execution of the program must be issued before you can enter a START command.) When the GO option is in effect, the compiled program will be loaded and executed automatically without any additional CMS commands. (FILEDEF commands must be issued before compilation.)

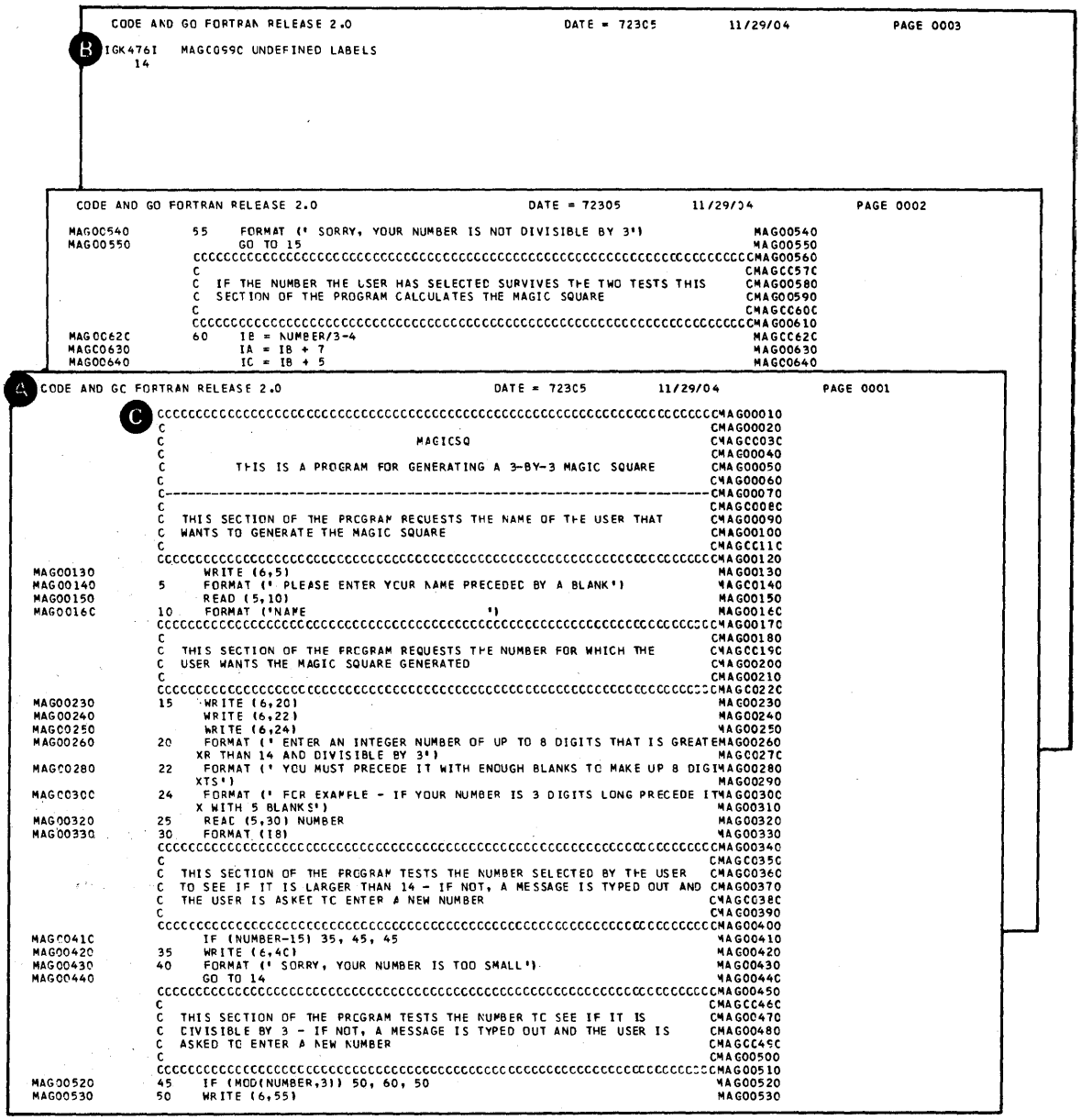


Figure 36. Code and Go FORTRAN Compiler LISTING File (Default Options)

Obtaining a Punch Card Deck of Your Object Program

To have your TEXT file punched into a card deck, you must specify the DECK option with the GOFORT command and use the PUNCH command.

Example:

```
punch newprog text
```

Compiler Language Restrictions for Code and Go FORTRAN

The language restrictions for Code and Go FORTRAN are as follows:

- The maximum number of arithmetic expressions that can be nested is 30 in a minimum CMS configuration. This limit increases as additional storage is made available.
- The maximum level of nested references from within an arithmetic function definition statement to another statement function or function subprogram is 10 in a minimum CMS configuration. This limit increases as additional storage is made available.
- The maximum number of source statements for one compilation is dependent upon the amount of storage available to the compiler. A minimum CMS configuration will allow up to a maximum of 230 statements or the equivalent.
- There is no restriction on the number of comments or connective comments in the source program. The maximum size of a statement is 1320 bytes exclusive of labels or sequence numbers but including any embedded blanks. This is equivalent to 19 fixed form continuation cards to a statement.
- The repetition field (a) for format codes in a FORMAT statement if present, must be an unsigned integer constant less than 256.
- The FORMAT statement specification w, indicating the number of characters of data in the field, must be an unsigned integer constant less than 256.
- In literal constants in the source program, any valid card code is permissible, except a 12-11-0-7-8-9 punch.
- In free form source, literal constants may be continued on a new line if the continuation begins in the first position of the new line.
- In the format statement no separator is required between H type format code strings or between X format codes and H format codes.

FORTRAN IV (H Extended) Compiler

Before you can use the H Extended compiler under CMS you must make sure that sufficient storage is available for it. The compiler requires a minimum of 600K bytes of storage, which is sufficient to compile small programs. You must issue a DEFINE STORAGE command specifying at least 600K prior to entering the IPL CMS command. The format of the DEFINE command is as follows:

DEFINE STORAGE *nnn*K

where:

nnn must be a minimum of 600. This value may be increased, in multiples of 4 up to the maximum allowed by CMS. The letter K represents 1024 bytes.

The compiler statistics can be used as a guide in calculating the most economical value for *nnn* (see part A of Figure 40). It lists the amount of space a particular program requires. As a guide a value of 800 will permit compilation of very large programs.

The FORTRAN command invokes the IBM FORTRAN IV (H Extended) compiler, which will compile a FORTRAN IV source program (see the publication *IBM System/360 and System/370 FORTRAN IV Language,* Order No. GC28-6515 for extensions to the FORTRAN IV language that the FORTRAN IV (H Extended) compiler will accept). Your source program must be contained in the CMS file that you identify in the command. You may include a set of options governing compiler operation and output; however, should you omit one or all of the options, defaults are assumed for you. If you include options that are not valid for the FORTRAN IV (H Extended) compiler or if you misspell any options a diagnostic message is typed out at your terminal (see "Appendix F" for more information). In the following illustrations and descriptions, all defaults for the compiler options are underlined.

FORTHX Command Format[1]

Figure 37 shows the format of the FORTHX command and the options that are available.

---

[1]The material in this section may be reproduced for internal use; it may not be offered for resale.

```
FORTHX    filename  ( [ALC|NOALC] [ANSF|NOANSF] [{AUTODBL|AD}(value)]
                    [BCD|{EBCDIC|EB}] [DECK|NODECK] [DISK|PRINT|NOPRINT]
                    [DUMP|NODUMP] [FLAG(I|E|S)]
                    [{FORMAT|FMT}|{NOFORMAT|NOFMT}] [GOSTMT|NOGOSTMT]
                    [{LINECOUNT|LC}(nn|60)] [LIST|NOLIST] [MAP|NOMAP]
                    [NAME( name|MAIN)] [{OBJECT|OBJ}|{NOOBJECT|NOOBJ}]
                    [{OPTIMIZE|OPT}(0|1|2)|{NOOPTIMIZE|NOOPT}]
                    [SIZE( nnnnK|MAX)] [{SOURCE|S}|{NOSOURCE|NOS}]
                    [TERM|NOTERM] [XREF|NOXREF] )
```

Note: If you specify more than 100 characters in the string of options, you will create an error condition and a message will be printed at your terminal. To correct this condition, reissue the command using abbreviations, where permitted, or specifying fewer options.

Figure 37. Format of the FORTHX Command for the FORTRAN IV (H Extended) Compiler

- Identifying the Compiler to be Used

  FORTHX -- The word FORTHX identifies the FORTRAN IV (H Extended) compiler and must be typed as shown.

- Specifying a File for Compilation

  filename -- Specifies the name of the file to be compiled. The file specified must have a filetype of FORTRAN or it will not be recognized by the FORTRAN IV (H Extended) compiler.

  Note: You must insure that the file named does not contain any statements that are not acceptable to the FORTRAN IV (H Extended) compiler (for example, free form source statements).

- Specifying Boundary Alignment of Data Items

  ALC -- Data items are to be aligned on proper storage boundaries. It may be used with the AUTODBL option to restore proper storage boundaries when a conversion is performed. (For more detailed information on the ALC option, see the section Automatic Precision Increase Facility).

  NOALC -- Data items will not be aligned on proper boundaries.

  If you omit this option, the compiler will assume NOALC.

- Library and Built-in Function Recognition

ANSF -- The compiler will recognize only those library and built-in functions specified by *American National Standard, (ANS) FORTRAN, X3.9-1966*. See the table of functions shown in the publication *IBM System/360 and System/370 FORTRAN IV Language,* Order No. GC28-6515 for a list of the ANS library and built-in functions. When this option is specified, any function that is not supported by ANS is assumed to be supplied by the user.

NOANSF -- The compiler will recognize the entire range of IBM-supplied library and built-in functions that are listed in the FORTRAN IV language manual.

If you omit this option, the compiler assumes NOANSF.

- Using the Automatic Precision Increase Facility

AUTODBL (*value*) -- The compiler will call the Automatic Precision Increase (API) facility. See the section "Automatic Precision Increase Facility" for more detailed information on the AUTODBL option. This option can be abbreviated AD(*value.*)

If you omit this option, the compiler will not perform any precision increase.

- Character Code of the Source Program

BCD -- The source program to be compiled is written in BCD.

Note: The CMS COPYFILE command with the EBCDIC option can be used to convert a file containing BCD code to a file in EBCDIC, thus eliminating the need for this option.

EBCDIC -- The source program to be compiled is written in EBCDIC code. This option can be abbreviated EB. If you omit this option, the compiler assumes EBCDIC.

- Producing a Card Deck for Your TEXT File

DECK -- The executable code produced by the compiler will be punched into a card deck in your computing center.

NODECK -- The executable code produced by the compiler will not be punched into a card deck.

If you omit this option, the compiler assumes NODECK.

- Producing a LISTING File for Your Program

DISK -- The compiler will place a copy of your LISTING file on a disk.

PRINT -- The compiler will print your LISTING file on the offline printer.

NOPRINT -- No LISTING file will be produced.

If you omit this operand, the compiler assumes DISK.

- Requesting a Dump

  DUMP -- The contents of registers, storage, and the files associated with the compiler are to be printed if an abnormal termination occurs.

  NODUMP -- No dump will be produced if an abnormal termination occurs.

  If you omit this option, the compiler assumes NODUMP.

- Specifying the Level of Diagnostic Messages to be Printed

  FLAG(*level*) -- Diagnostic messages, of the *level* indicated will be printed at your terminal and included in the LISTING file. A level of I indicates that informative 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) are to be printed. A level of E indicates that only error messages and severe error messages are to be printed. A level of S indicates that only severe error messages are to be printed.

  If you omit this option, the compiler assumes FLAG (I).

- Producing a Structured Source Program

  FORMAT -- A structured source program listing indicating the loop structure and logical continuity of your source program is included in the LISTING file produced by the compiler. This option can be abbreviated FMT.

  Note: This option is useful only when the OPTIMIZE 2 option is in effect.

  NOFORMAT -- A structured source program listing will not be produced. This option can be abbreviated NOFMT.

  If you omit this option, the compiler assumes NOFORMAT.

- Generating Internal Statement Numbers

  GOSTMT -- Internal Sequence Numbers (ISN) are to be generated for the calling sequence to subroutines for a traceback map.

  NOGOSTMT -- Internal Statement Numbers will not be generated.

  If you omit this option, the compiler assumes NOGOSTMT.

- Number of Lines to be printed on Each Listing Page

  LINECOUNT(*nn*) -- The source listing for your program will be printed with a maximum of *nn* lines per page. You may specify any number for *nn* from 1 to 99. This option can be abbreviated LC.

  If you omit this option, the compiler assumes a line count of 60.

- Producing a Listing of Your Object Module

LIST -- The compiler will include, in the LISTING file, a pseudo-assembler listing of the translated statements contained in the TEXT file.

NOLIST -- The pseudo-assembler listing for your program will not be included in the LISTING file.

If you omit this option, the compiler assumes NOLIST.

- Producing a Variable and Label Map for Your Source Program

MAP -- The compiler will generate a table of variable names and statement labels. This table will be included in your LISTING file.

NOMAP -- The compiler will not generate tables for variable names and statement labels.

If you omit this option, the compiler will assume NOMAP.

- Naming Your Program

NAME (*name*) -- The name represented by *name* will be assigned by the compiler to the executable code it produces. You may specify from one to 6 characters for *name*.

If you omit this option, the compiler assigns the name MAIN to your executable code.

- Producing an Object Module

OBJECT -- The compiler will create executable code from the FORTRAN source code in your program. This code will be placed in the TEXT file. This option can be abbreviated OBJ.

NOOBJECT -- The compiler will not produce executable code or a TEXT file. This option can be abbreviated NOOBJ.

If you omit this option, the compiler assumes OBJECT.

- Specifying the Level of Optimization for Your Compilation

OPTIMIZE (*level*) -- The compiler will perform the type of optimization indicated by the *level*. A level of 0 indicates that no optimization is to be performed. A level of 1 indicates that each source module is to be treated as a single program loop and is to be optimized without regard for register allocation or branching. A level of 2 indicates that each source module is to be treated as a collection of program loops and that each loop is to be optimized with regard for register allocation, branching, common expression elimination, and replacement of redundant computations. Optimizing techniques are discussed further in the "Programming Considerations" sections. This option can be abbreviated OPT.

NOOPTIMIZE -- The compiler will not perform any optimization. This option is equivalent to specifying OPTIMIZE (0). This option can be abbreviated NOOPT.

If you omit this option, the compiler assumes NOOPTIMIZE.

- Specifying the Amount of Main Storage for Your Compilation.

  SIZE (MAX) -- The compiler will use all available storage, except for approximately 3K bytes, which are left for system routines.

  SIZE (*nnnn*K) -- The amount of storage occupied and used by the compiler is limited to the value indicated by (*nnnn*K). The value of *nnnn* can be any number from 460 to 9999. This capability is not intended for normal compiler operation. Its use should be restricted to only those applications in which a problem program invokes the compiler through a CALL, ATTACH, or LINK macro instruction and, therefore, must limit the amount of storage available to the compiler. As a guide to establishing a SIZE, remember that the compiler is usually loaded into storage beginning at 128K. The length of the compiler is 460K plus the amount of storage occupied by the CMAJOR and ADCON compiler tables. In addition, the compiler requires at least 12K of workspace. Your application will determine the exact amount of additional workspace that the compiler will use. The compiler diagnostics will indicate the amount of unused workspace. The SIZE you choose must account for the length of the compiler plus tables and workspace. Storage at addresses higher than SIZE is available for your application. This storage may be extended with the DEFINE STORAGE command. Be aware that limiting the amount of storage available to the compiler may adversely affect its performance.

  If you omit this option, the compiler assumes SIZE (MAX).

- Producing a Listing of Your Source Program

  SOURCE -- The compiler will include a copy of your source program in the LISTING file it produces for your program. This option can be abbreviated S.

  NOSOURCE -- A copy of your source program will not be included in the LISTING file. This option can be abbreviated NOS.

  If you omit this option, the compiler assumes SOURCE.

- Typing Compiler Error Messages at Your Terminal

  TERM -- Any erroneous statements detected in your FORTRAN program and the corresponding messages will be typed at your terminal.

  NOTERM -- Errors and messages will not be typed at your terminal.

  If you omit this operand, the compiler assumes TERM.

- Producing a Cross-Reference Listing of Variables and Labels

  XREF -- A cross-reference listing of variable names and labels used in your program will be included in the LISTING file produced for your program. If XREF is specified, ISNs are generated (regardless of whether GOSTMT was specified) for each statement in which a variable or label was used.

NOXREF -- A cross-reference listing will not be included in your FORTRAN IV (H Extended) LISTING file.

If you omit this option, the compiler assumes NOXREF.

*Changing Compiler Options with a *PROCESS Statement*

The H Extended compiler permits a source program to set the compiler options that it will require regardless of the defaults or the options that you specified in the FORTHX command. The compiler accepts a *PROCESS statement which may contain any of the compiler options that you want to use in place of the corresponding compiler defaults. This facility permits you to specify a different set of options for each source program in a file that contains more than one source program. The options used will be either the compiler defaults or the options specified in the *PROCESS statement. You do not need to place a *PROCESS statement in the first program since its options are set by the FORTHX command.

To code a *PROCESS statement, type an asterisk (*) in column 1 (starting at the left hand margin indicator); type the word PROCESS in columns 2 through 8, and leave column 9 blank. You may place the compiler options that you want anywhere after column 9 but before column 72, which must be left blank indicating the end of the statement. When used in a source program, the *PROCESS statement must be the first statement in the program.

Example:

```
*process deck, optimize(2)
```

You may use all the options shown in Figure 37 except SIZE, DISK, PRINT, or NOPRINT.

Output from the FORTRAN IV (H Extended) Compiler

The FORTRAN IV (H Extended) compiler produces a LISTING file that contains any errors detected during compilation and diagnostic messages. It may also include a copy of your source program, a map of variable names and labels, a cross-reference list of variable names and labels, a pseudo-assembler listing of the executable code produced, a dump in the event of abnormal termination, an edited source program, and internal statement numbers in the source program. In addition, all erroneous statements and diagnostic messages included in the LISTING file will be printed at your terminal. A second file, the TEXT file, can also be produced that will contain the actual executable code. The FORTRAN IV (H Extended) compiler can also produce a punched card form of your TEXT file. See Figure 38 for a summary of the FORTRAN IV (H Extended) compiler options and their effect on output.

Option	LISTING File	Text File	Ternminal Response
SOURCE	Includes the source code from the CMS FORTRAN source file		
OBJECT		Creates this file	
XREF	Includes a cross reference list of variables and labels		
LIST	Includes a pseudo-assembler listing of the executable code produced		
FORMAT	Includes an edited copy of the source code		
MAP	Includes address tables for variables and labels		
DECK		Punches a copy of this file off line	
PRINT	Creates this file prints a copy offline		
DISK	Creates this file and writes a copy on an available disk		
TERM			Prints all error and diagnostic messages at the terminal

Figure 38. The Effect of Various Compiler Options on Compiler Output

FORTRAN IV (H Extended) LISTING File

The LISTING file is always produced by the FORTRAN IV (H Extended) compiler unless the NOPRINT option is in effect. It contains the following:

- Informative messages that indicate the status of the compilation.
- Any errors detected during the compilation and the corresponding diagnostic messages. For a detailed description of the diagnostic messages produced by the FORTRAN IV (H Extended) compiler refer to the publication *IBM System/360 Operating System: FORTRAN IV (H Extended) Compiler and Library (Mod II) Messages,* Order No. SC28-6885.

- Optional output as determined by the options you can specify with the FORTHX command or their defaults.

Should you need to edit the LISTING file you must include the option (LRECL 133) with your EDIT command.

Informative Messages

The informative messages included in your listing identifies the compiler used, the Julian date, and the time of day (based on a 24-hour clock) that the compilation was begun. A list of the compiler options requested, all the options that were in effect, and the compiler statistics are also provided. See the part of Figure 40 labeled A for an illustration of the compiler informative messages.

Error Messages

The error messages produced by the FORTRAN IV (H Extended) compiler are listed in a group. They contain the error number (IFE*xxx* I), its severity level, and the text of the message. When the message refers to a specific source statement, an internal sequence number is included to identify the statement in error.

Messages with a severity level of 4 permit you to execute your program. Severity levels higher than 4 prevent execution from taking place. See the part of Figure 40 labeled B for an illustration of the compiler error messages.

Optional Output

Additional information can be included in your LISTING file, depending on the defaults in effect and the options you specify with the FORTHX command. You can include the following:

- A list of your source statements.
- A source program map.
- An edited list of your source program.
- A cross-reference listing of your source program.
- A list of the object code produced for your source program.

Obtaining a Copy of Your Source Statements

Since the SOURCE option is the default for the FORTHX command, you will have a copy of your source program included in your LISTING file automatically. See the part of Figure 40 labeled C for the format of the source listing. If you do not want your source statements included, you must specify the NOSOURCE option.

Obtaining a Source Module Map

The first part of the source module map is a table that contains entries generated by the compiler for each of eleven classifications of variables that you may have used in your program. The first line of the map gives the name of the program and its size in hexadecimal format. The column labeled TAG

indicates the classification of each variable. Figure 39 explains the classifications.

Classification	Meaning
A	A variable that was used as an argument in a parameter list
ASF	An arithmetic statement function
C	A variable that appeared in a COMMON block
D	A promoted (doubled) variable
E	A variable that appeared in an EQUIVALENCE block
F	A variable that appeared to the right of an equal sign (that is, a variable whose value was manipulated during some operation)
P	A padded variable
S	A variable that appeared to the left of an equal sign (that is, a variable whose value was stored during some operation)
XF	An external function
XR	An external reference to an array or a variable
*	A promoted library function
Note: The combination code ASF should not be confused with the individual A, S, and F. When a variable has been used for these several purposes, the individual codes will appear as SFS to avoid confusing it with the arithmetic function code is always ASF.	

Figure 39. H Extended Storage Map Variable Classifications

The column labeled TYPE indicates the type and length of each variable listed.

The column labeled ADD indicated the relative address assigned to the variable name. (Functions, arithmetic statement functions, subroutines, and external references have a relative address of 00000.) For variables that you have not referred to, the letters NR will appear instead of a relative address.

The second part of the source module map is a table of statement numbers. This label map shows each statement number that you used in your source program and any labels that the compiler generated. The relative address assigned to each label is also shown. Any unreferenced symbols are indicated by the letters NR instead of a relative address.

If the source module contains COMMON or EQUIVALENCE statements, a third part of the source module map is included. The map for COMMON blocks contains the same kind of information as for the main program. Any variable that is made equivalent to a variable in a COMMON block is listed along with its displacement (offset) from the beginning of the block. See the part of Figure 40 labeled D for the format of the source module map. If you want the map included in your LISTING file, you must specify the MAP option with the FORTHX command.

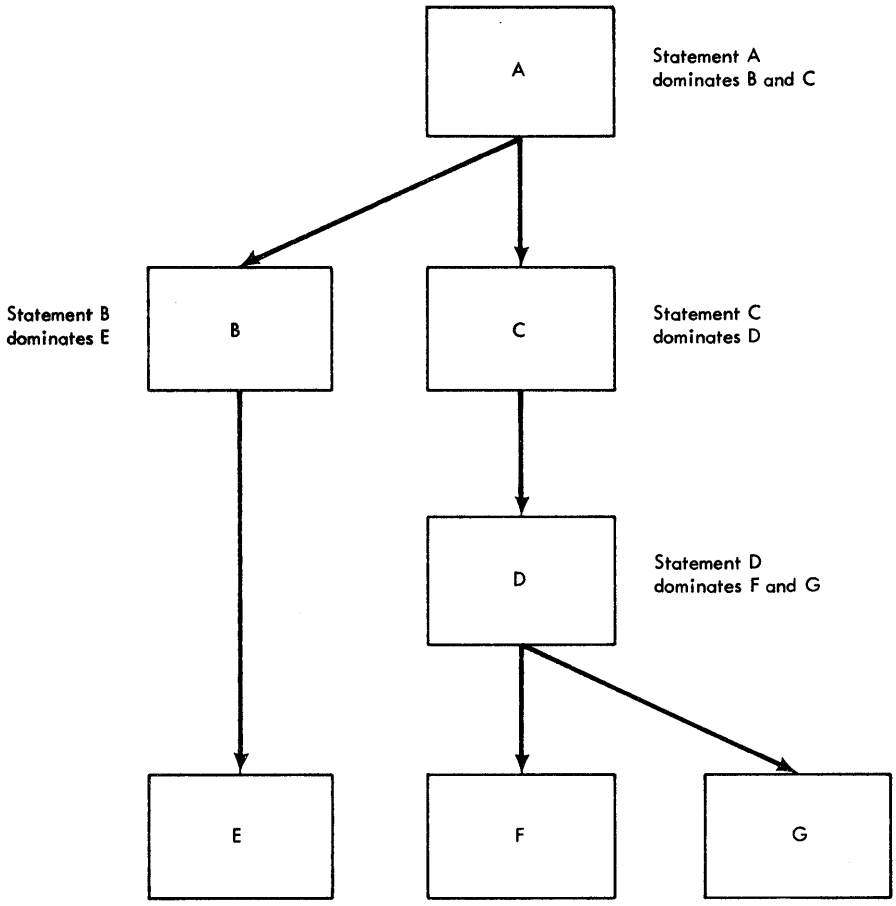
Obtaining an Edited Copy of Your Source Program

The edited copy of your source program is independent of the usual source listing; it indicates the loop structure and logical continuity of the program.

Each loop in your program is assigned a unique 3-digit number. The entrance to each loop is indicated by a left parenthesis followed by a 3-digit number; the exit from that loop is indicated by the same 3-digit number followed by a right parenthesis.

The logical continuity of your program is shown through the dominance relationships among executable source statements. A statement dominates another if *all* logical paths to the second statement go through the first. The first statement is called the dominator and the second is called the dominee. By this definition, a statement can have only one dominator, but a dominator may have several dominees. For example, a computed GO TO statement is the last statement through which control passes before reaching three other statements. The GO TO statement is a dominator with three dominees.

Example:



In the listing, a dominee is indentend from its dominator unless it is the only dominee or the last dominee of that dominator. The indention may be broken by intervening statements; this is dominance discontinuity and is indicated by a C--- on a separate line above the dominee.

Comments and non-executable statements are not involved in dominance relationships. Their presence never causes a dominance discontinuity. Comments are aligned with the last preceding non-comment line and non-executable statements are aligned with the last preceding executable statement or the first one following. See the part of Figure 40 labeled E for the format of the edited source listing. If you want a copy included in the LISTING file, you must specify the FORMAT and OPTIMIZE2 options with the FORTHX commannd.

Obtaining a Cross-reference Listing

The cross-reference listing shows the symbols and statement labels that you used in your source program with the internal sequence numbers of the statements in which they appeared. Symbols, which define variables, are listed by name, alphabetically. Statement labels are listed in numeric sequence. The internal sequence number of the statements that define and reference each symbol and label are shown after them. See the part of Figure 40 lableled F for the format of the cross-reference listing. If you want a copy of the listing included in the LISTING file, you must specify the XREF option with the FORTHX command.

Obtaining a Copy of Your Pseudo-assembler Listing of Your Executable Code

The pseudo-assembler listing contains your FORTRAN source statements after they have been translated into an executable form by the FORTRAN IV (H Extended) compiler. This listing represents the executable code in an assembler language format. The following items are shown:

- The column labeled 1 indicates the relative address (in hexadecimal format) of the assembler language instruction.
- The column labeled 2 indicates the storage representation (in hexadecimal format) of the instruction.
- The column labeled 3 indicates the statement numbers you used in your program or which the compiler generated (6-digit numbers).
- The column labeled 4 indicates the pseudo-assembler language code for each statement.
- The column labeled 5 indicates any significant items referred to by the instruction, such as, entry points of subroutines or other statement numbers.

See the part of Figure 40 labeled G for the format of the object code listing. If you want a copy of this listing included in the LISTING file, you must specify the LIST option with the FORTHX command.

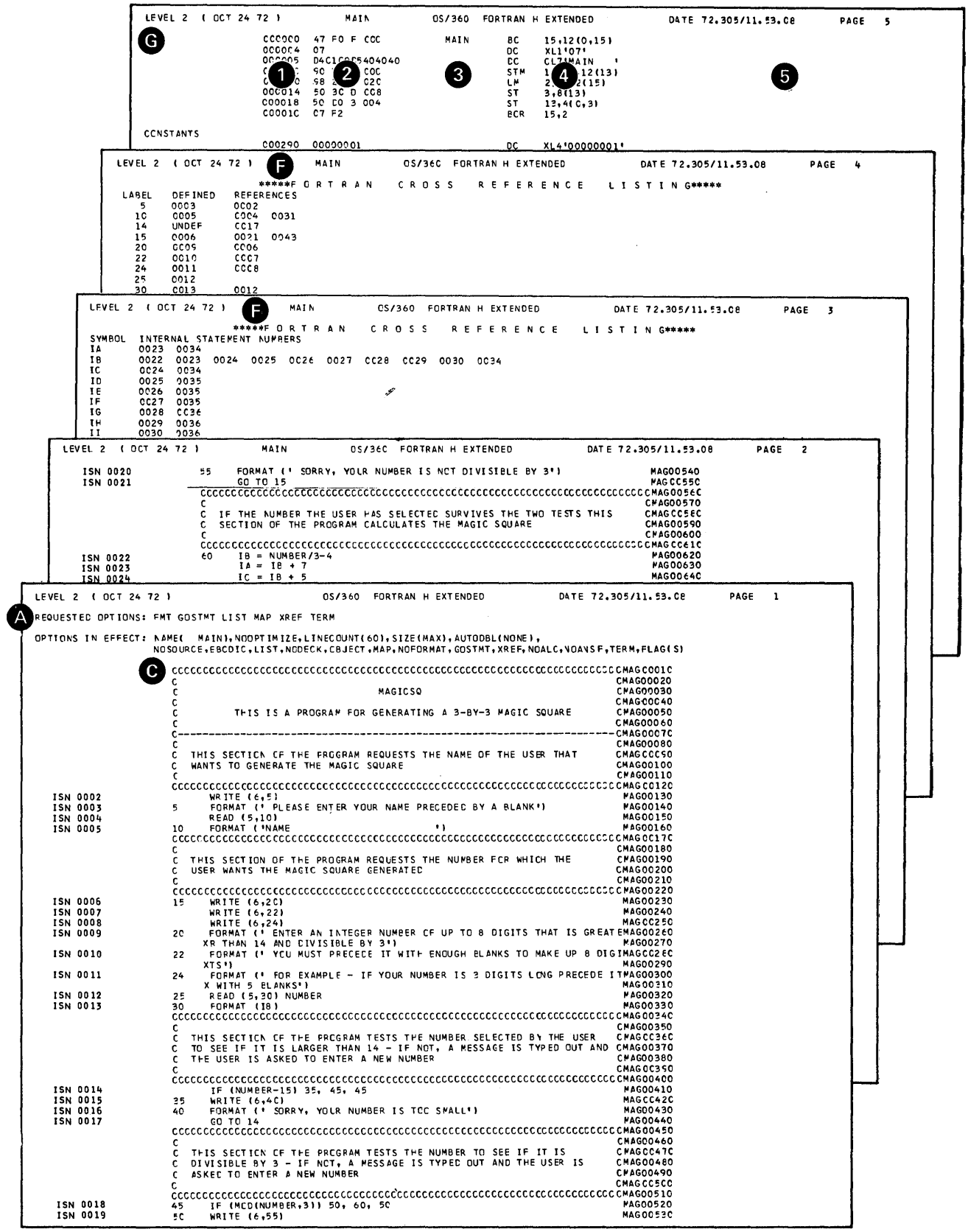
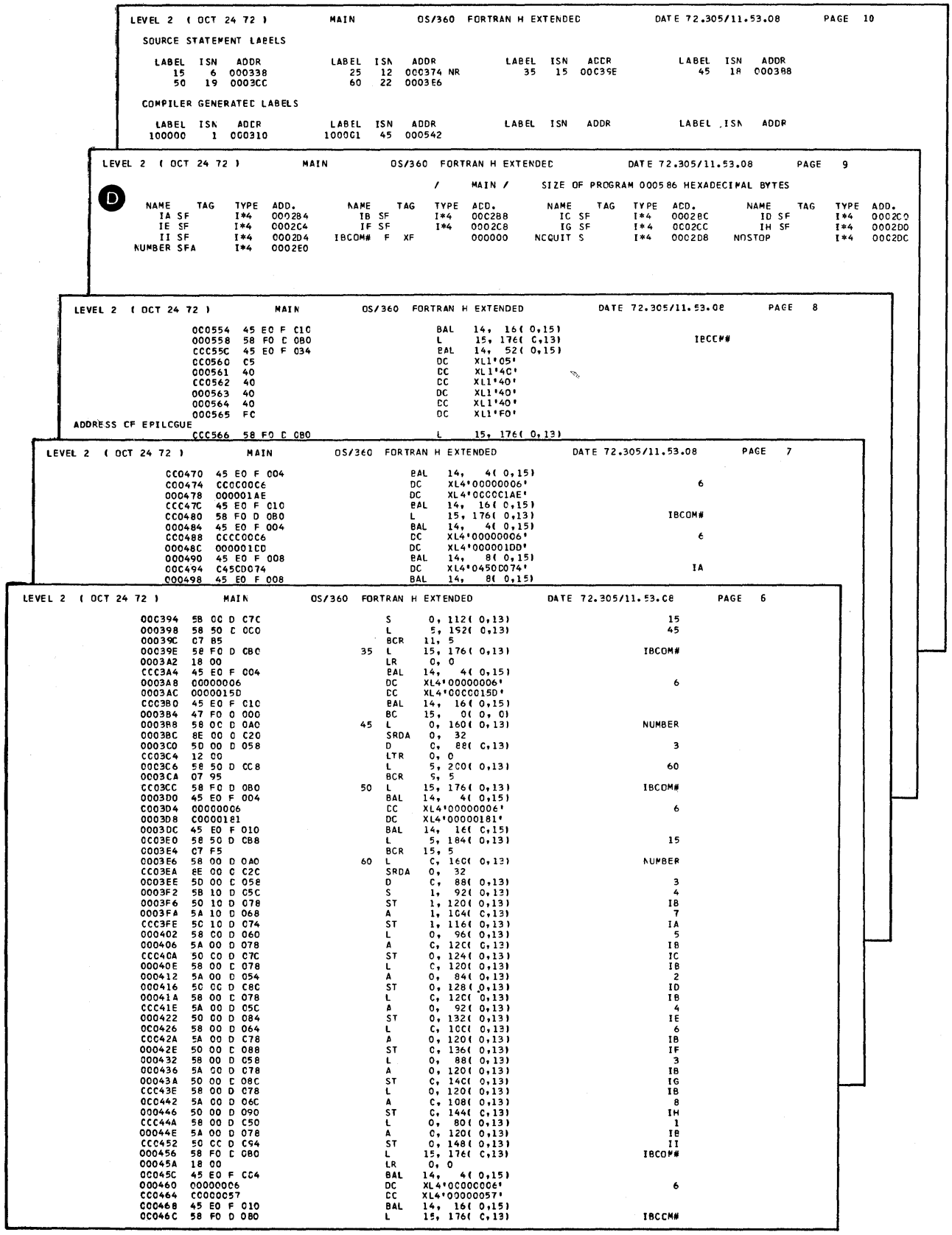


Figure 40. FORTRAN IV (H Extended) LISTING File (Part 1 of 3)



```
LEVEL 2  ( OCT 24 72 )          MAIN          OS/360  FORTRAN H EXTENDED            DATE 72.305/11.53.08       PAGE  10

  SOURCE STATEMENT LABELS

    LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR
       15    6  000338             25   12  000374 NR          35   15  00C39E             45   18  000388
       50   19  0003CC             60   22  0003E6

  COMPILER GENERATED LABELS

    LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR          LABEL  ISN   ADDR
   100000    1  000310         100001   45  000542
```

```
LEVEL 2  ( OCT 24 72 )          MAIN          OS/360  FORTRAN H EXTENDED            DATE 72.305/11.53.08       PAGE   9

                                                /     MAIN /     SIZE OF PROGRAM 000586 HEXADECIMAL BYTES
(D)
    NAME    TAG   TYPE  ADD.        NAME    TAG   TYPE  ADD.        NAME    TAG   TYPE  ADD.        NAME    TAG   TYPE  ADD.
      IA SF       I*4   000284        IB SF       I*4   00C2B8        IC SF       I*4   00028C        ID SF       I*4   0002C0
      IE SF       I*4   0002C4        IF SF       I*4   0002C8        IG SF       I*4   0002CC        IH SF       I*4   0002D0
      II SF       I*4   0002D4    IBCOM#  F  XF          000000    NCQUIT S        I*4   0002D8    NOSTOP          I*4   0002DC
  NUMBER SFA      I*4   0002E0
```

```
LEVEL 2  ( OCT 24 72 )          MAIN          OS/360  FORTRAN H EXTENDED            DATE 72.305/11.53.08       PAGE   8

                00C0554  45 E0 F C10                    BAL   14,  16( 0,15)
                000558  58 F0 C 0B0                     L     15, 176( C,13)                IBCOM#
                00C55C  45 E0 F 034                     BAL   14,  52( 0,15)
                00C560  C5                              DC    XL1'05'
                000561  40                              DC    XL1'4C'
                00C562  40                              DC    XL1'40'
                000563  40                              DC    XL1'40'
                000564  40                              DC    XL1'40'
                000565  FC                              DC    XL1'FC'
ADDRESS OF EPILOGUE
                00C566  58 F0 C 0B0                     L     15, 176( 0,13)
```

```
LEVEL 2  ( OCT 24 72 )          MAIN          OS/360  FORTRAN H EXTENDED            DATE 72.305/11.53.08       PAGE   7

                00C0470  45 E0 F 004                    BAL   14,   4( 0,15)
                00C0474  00C00006                       DC    XL4'00000006'                  6
                000478  000001AE                       DC    XL4'00C00C1AE'
                00C47C  45 E0 F 010                     BAL   14,  16( 0,15)
                00C0480  58 F0 D 0B0                    L     15, 176( 0,13)               IBCOM#
                000484  45 E0 F 004                     BAL   14,   4( 0,15)
                00C0488  00C00006                       DC    XL4'00000006'                  6
                00048C  000001DD                        DC    XL4'000001DD'
                000490  45 E0 F 008                     BAL   14,   8( 0,15)
                00C0494  C45CD074                       DC    XL4'0450D074'                 IA
                000498  45 E0 F 008                     BAL   14,   8( 0,15)
```

```
LEVEL 2  ( OCT 24 72 )          MAIN          OS/360  FORTRAN H EXTENDED            DATE 72.305/11.53.08       PAGE   6

                00C394  5B 0C D C7C                    S      0, 112( 0,13)                 15
                000398  58 50 C 0C0                    L      5, 192( 0,13)                 45
                00039C  07 B5                          BCR    11, 5
                00039E  58 F0 D CB0             35     L      15, 176( 0,13)              IBCOM#
                0003A2  18 00                          LR     0, 0
                00C3A4  45 E0 F 004                    BAL    14,   4( 0,15)
                0003A8  00000006                       DC     XL4'00000006'                  6
                0003AC  0000015D                       DC     XL4'00CC015D'
                00C3B0  45 E0 F C10                    BAL    14,  16( 0,15)
                0003B4  47 F0 0 000                    BC     15,   0( 0, 0)
                0003B8  58 0C D 0A0             45     L      0, 160( 0,13)               NUMBER
                0003BC  8E 00 0 C20                    SRDA   0,  32
                0003C0  5D 00 D 058                    D      0,  88( 0,13)                  3
                00C3C4  12 00                          LTR    0, 0
                00C3C6  58 50 D CC8                    L      5, 200( 0,13)                 60
                0003CA  07 95                          BCR    9, 5
                00C3CC  58 F0 D 0B0             50     L      15, 176( 0,13)              IBCOM#
                0003D0  45 E0 F 004                    BAL    14,   4( 0,15)
                00C3D4  00000006                       DC     XL4'00000006'                  6
                0003D8  C0000181                       DC     XL4'00000181'
                0003DC  45 E0 F 010                    BAL    14,  16( C,15)
                00C3E0  58 50 D CB8                    L      5, 184( 0,13)                 15
                00C3E4  07 F5                          BCR    15, 5
                0003E6  58 00 D 0A0             60     L      C, 160( 0,13)               NUMBER
                00C3EA  8E 00 C C2C                    SRDA   0,  32
                0003EE  5D 00 C 058                    D      C,  88( 0,13)                  3
                0003F2  5B 10 D C5C                    S      1,  92( 0,13)                  4
                0003F6  50 10 D 078                    ST     1, 120( 0,13)                 IB
                0003FA  5A 10 D 068                    A      1, 104( C,13)                  7
                00C3FE  5C 10 D 074                    ST     1, 116( 0,13)                 IA
                000402  58 C0 D 060                    L      0,  96( 0,13)                  5
                000406  5A 00 D 078                    A      C, 120( C,13)                 IB
                00C40A  50 C0 D C7C                    ST     0, 124( 0,13)                 IC
                00040E  58 00 C 078                    L      C, 120( 0,13)                 IB
                000412  5A 00 D 054                    A      0,  84( 0,13)                  2
                000416  50 CC D C8C                    ST     0, 128( 0,13)                 ID
                00041A  58 00 C 078                    L      C, 120( 0,13)                 IB
                00C41E  5A 00 D 05C                    A      0,  92( 0,13)                  4
                000422  50 00 D 084                    ST     0, 132( 0,13)                 IE
                00C426  58 00 D 064                    L      C, 100( 0,13)                  6
                00C42A  5A 00 D C78                    A      0, 120( 0,13)                 IB
                00042E  50 00 C 088                    ST     C, 136( 0,13)                 IF
                000432  58 00 D 058                    L      0,  88( 0,13)                  3
                000436  5A 00 D C78                    A      0, 120( 0,13)                 IB
                00043A  50 00 C 08C                    ST     C, 140( 0,13)                 IG
                00C43E  58 00 D C78                    L      0, 120( 0,13)                 IB
                00C442  5A 00 D 06C                    A      C, 108( 0,13)                  8
                000446  50 00 D 090                    ST     C, 144( C,13)                 IH
                00C44A  58 00 D C50                    L      0,  80( 0,13)                  1
                00044E  5A 00 D 078                    A      0, 120( 0,13)                 IB
                00C452  50 CC D C94                    ST     0, 148( 0,13)                 II
                000456  58 F0 C 0B0                    L      15, 176( C,13)              IBCOM#
                00045A  18 00                          LR     0, 0
                00C45C  45 E0 F CC4                    BAL    14,   4( 0,15)
                000460  00000006                       DC     XL4'0C000006'                  6
                00C464  C0000057                       DC     XL4'00000057'
                00C468  45 E0 F 010                    BAL    14,  16( 0,15)
                00C46C  58 F0 D 0B0                    L      15, 176( C,13)              IBCOM#
```

Figure 40. FORTRAN IV (H Extended) LISTING File (Part 2 of 3)

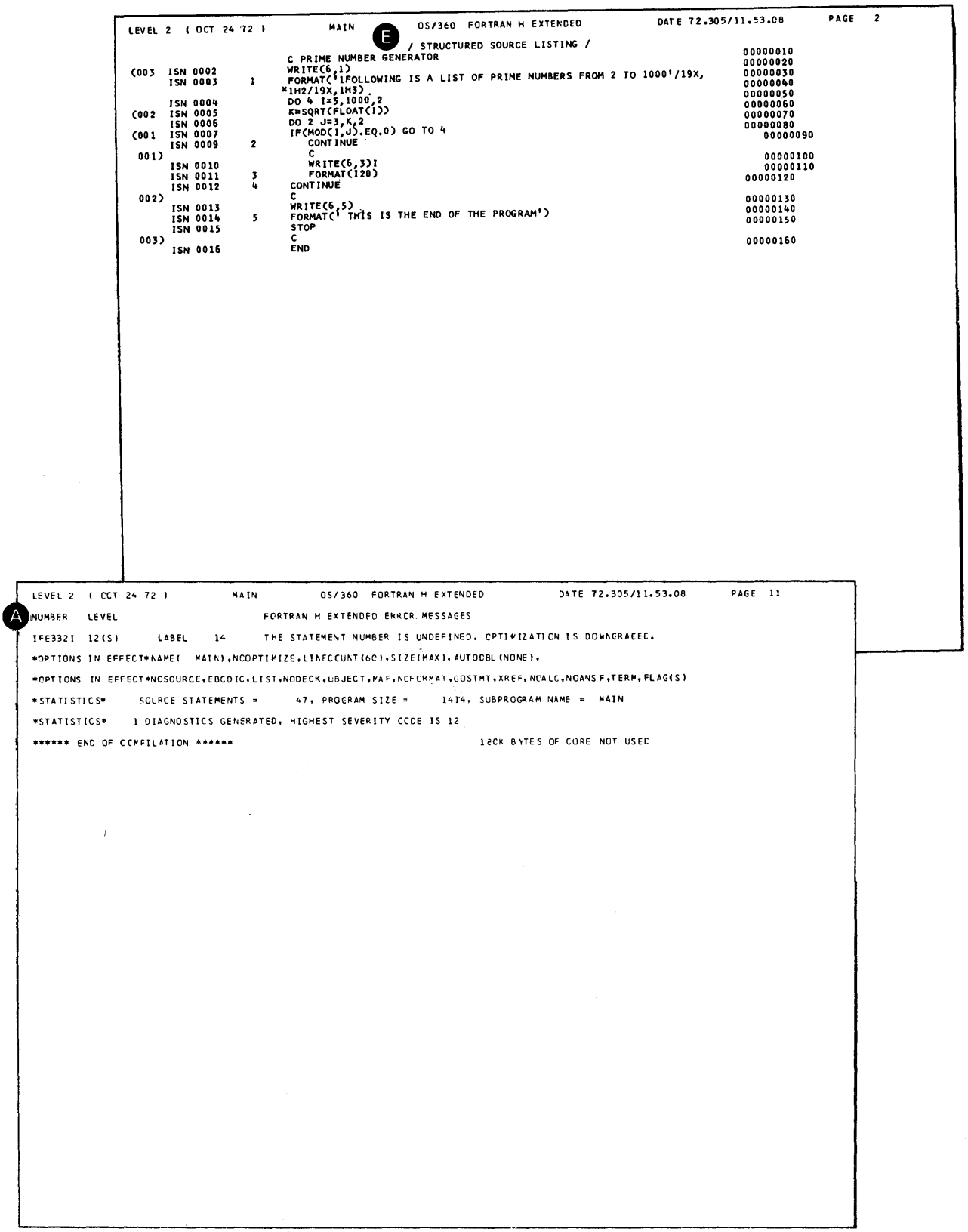


Figure 40. FORTRAN IV (H Extended) LISTING File (Part 3 of 3)

FORTRAN IV (H Extended) TEXT File

A TEXT file is created by the FORTRAN IV (H Extended) compiler whenever the OBJECT option is in effect. This file contains the compiled FORTRAN IV object program that is identical to the object program produced by the FORTRAN IV (H Extended) compiler under OS.

The TEXT file contains the version of your program that can be executed by CMS or link edited under OS. You can issue a RUN command, LOAD and a START command, or an EXEC command that specifies a file containing these commands and your program will begin executing. (Any additional FILEDEF commands required for the execution of the program must be issued before you can enter a START command.)

Obtaining a Punch Card Deck of Your Object Program

To have your object program punched into a card deck, you must specify the DECK option with the FORTHX command. An alternate method is to use the PUNCH command specifying the filename of your file and a filetype of TEXT. This will punch the TEXT file that was created and placed on a disk by the OBJECT option when specified during a compilation.

Example:

```
punch newprog text
```

Compiler Language Restrictions for FORTRAN IV (H Extended)

The compiler language restrictions for the FORTRAN IV (H Extended) compiler are as follows:

- The maximum number of nested open DO statements is 25.
- The maximum number of implied DOs per input/output statement is 20.
- The maximum value for a repetition field (a) in a FORMAT statement is 255.
- The maximum value for the character specification field (w) in a FORMAT statement is 255.
- The maximum number of arguments in a statement function definition is 20.
- Within a statement function definition, the maximum number of nested references to other statement functions is 50.
- Within a statement function reference, the maximum number of nested references to other statement functions is 50.

- The maximum number of arguments in a CALL statement is 196; any argument containing a subscript is counted as two.

- The maximum number of characters permitted in a PAUSE statement is 255.

- The maximum number of characters permitted in literal constants is 255; this restriction applies to literal constants specified in list-directed input and output statements (statements with no corresponding FORMAT statement).

- The asynchronous input and output facility is not available under CMS. This feature is designed to make input and output more efficient under OS, in a batch environment. Since input and output are handled differently under CMS, in a time-sharing environment, programs using this feature may be compiled but not executed under CMS.

  If you attempt to load and execute a program that uses asynchronous I/O, the message:

  THE FOLLOWING NAMES ARE UNDEFINED:

  IN#

  OUT#

  WAIT#

  will be produced, indicating the type of asynchrounous operation specified.

- The compiler options SIZE , DISK, PRINT, and NOPRINT may not be specified in an *PROCESS statement.

Loading and Executing FORTRAN Object Programs Under CMS

Once a TEXT file has been created by your compiler, you are ready to load and execute it (unless, you are using the Code and Go compiler with the GO option, which will load and execute the object program automatically after compilation). The following examples illustrate CMS command procedures for compiling a FORTRAN source program, loading the resultant TEXT file, and executing it using the following compilers:

- FORTRAN IV (G1)
- Code and Go FORTRAN IV (with the GO option)
- Code and Go FORTRAN IV (with the NOGO option)
- FORTRAN IV (H Extended)

The commands shown can be placed in an EXEC procedure that will automatically perform all the functions whenever the name of the procedure is entered. See the publication *IBM VM/370 Command Language User's Guide*, Order No. GC20-1804 for more detailed information on preparing EXEC procedures and supplying filenames as arguments.

Command Procedure for FORTRAN IV (G1)

Example:

```
1  fortgi filename ( load )
     .
     .
     .
   R;

2  global txtlib library names
   R;

3  load filename
   R;

4  filedef ftxxfyyy device
5  start main
     .
     .
     .
   R;
```

An explanation of the numbered statements follows:

1. Supply the name of the file that contains the FORTRAN source program you want to compile. The FORTGI command invokes the FORTRAN IV (G1) compiler. The LOAD option specifies that a TEXT file is to be created.

2. If you do not have a PROFILE EXEC procedure with a GLOBAL TXTLIB command for the files that contain the Mod I library, enter a GLOBAL TXTLIB command here specifying the names your installation has assigned to the Mod I Library.

3. Specify the filename that you used in step 1. The LOAD command invokes the CMS loader, which loads the TEXT file produced by the compiler and prepares it for execution. The CMS loader produces a MAP file that contains the names of the modules loaded and the locations at which they were loaded. You can use the TYPE or PRINT commands, specifying LOADMAP or use the TMAP and TYPE options of the LOAD command to obtain a copy for use as a debugging aid.

4. If your program requires any user-defined execution-time files, issue the FILEDEF commands for them at this point. See the section of this book "User-defined Files" for more information.

5. Specify the default name MAIN that is assigned by the compiler unless you specified a name with the NAME option (see the section "Identifying Programs in a TEXT File" for more information) and execution of your program begins.

Command Procedure for Code and Go FORTRAN IV (with the GO Option)

Example:

```
1  global txtlib tsolib library  names
   R;

2  filedef ftxxfyyy device
   R;

3  gofort filename (GO)
   .
   .
   .
   R;
```

An explanation of the numbered statements follows:

1. If you do not have a PROFILE EXEC procedure with a GLOBAL TXTLIB command for the files that contain the Mod I library, enter a GLOBAL TXTLIB command here, specifying TSOLIB and the names that your installation has assigned to the Mod I Library. TSOLIB contains system routines necessary for input and output operations.

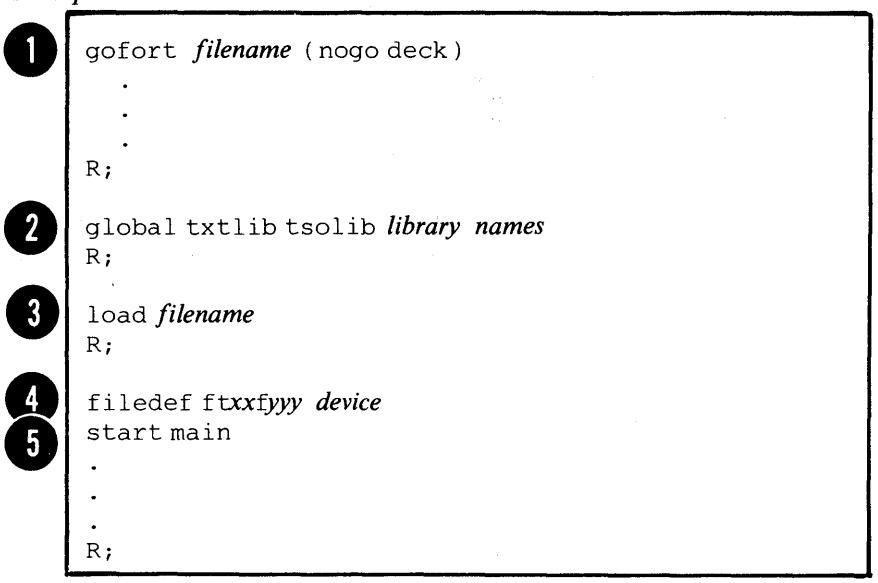
2. If your program requires any user-defined execution-time files, issue the FILEDEF commands for them at this point. See the section of this book "User-defined Files" for more information.

3. Supply the name of the file that contains the FORTRAN source program you want to compile. The GOFORT command invokes the Code and Go FORTRAN IV compiler. The GO option indicates that

you want the object code loaded and executed automatically after compilation. If the file that you named contains free-form source statements, be sure to include the FREE option.

Command Procedure for Code and Go FORTRAN IV (with the NOGO Option)

Example:



An explanation of the numbered statements follows:

1. Supply the name of the file that contains the FORTRAN source program you want to compile. The GOFORT command invokes the Code and Go FORTRAN IV compiler. The DECK option specifies that a TEXT file is to be created. The NOGO option indicates that you do not want the TEXT file loaded and executed automatically after compilation. If the file that you named contains free-form source statements, be sure to include the FREE option.

2. If you do not have a PROFILE EXEC procedure with a GLOBAL TXTLIB command for the files that contain the Mod I library, enter a GLOBAL TXTLIB command here specifying TSOLIB and the names that your installation has assigned to the Mod I Library. TSOLIB contains system routines necessary for input and output operations.

3. Specify the filename that you used in step 1. The LOAD command invokes the CMS loader, which loads the object code in the TEXT file produced by the compiler and prepares it for execution. The CMS loader produces a MAP file that contains the names of the modules loaded and the locations at which they were loaded. You can use the TYPE or PRINT commands, specifying LOAD MAP, or use the MAP and TYPE options of the LOAD command to obtain a copy for use as a debugging aid.

4. If your program requires any user-defined execution-time files, issue the FILEDEF commands for them at this point. See the section of this book "User-defined Files" for more information.

5. Specify the default name MAIN that is assigned by the compiler unless you specified a name with the NAME option (see the section "Identifying Programs in a TEXT File" for more information) and execution of your program begins.

Command Procedure for FORTRAN IV (H Extended)

Example:

```
1  forthx filename (object)
   R;

2  global txtlib cmslib library names
   R;

3  load filename
   R;

4  filedef ftxxfyyy device
5  start main
   .
   .
   .
   R;
```

An explanation of the numbered statements follows:

1. Supply the name of the file that contains the FORTRAN source program you want to compile. The FORTHX command invokes the FORTRAN IV (H Extended) compiler. The OBJECT option specifies that a TEXT file is to be created.

2. If you do not have a PROFILE EXEC procedure with a GLOBAL TXTLIB command for CMSLIB and the files that contain the Mod II library, enter a GLOBAL TXTLIB here, specifying CMSLIB and the name that your installation has assigned to the Mod II Library. CMSLIB contains the extended precision simulation routines.

   Note: You do not need to include CMSLIB if you are not using extended precision.

3. Specify the filename that you used in step 1. The LOAD command invokes the CMS loader, which loads the object code in the TEXT file produced by the compiler and prepares it for execution. The CMS loader produces a MAP file that contains the names of the modules loaded and the locations at which they were loaded. You can use the TYPE or PRINT command, specifying LOAD MAP, or use the MAP and TYPE options of the LOAD command to obtain a copy for use as a debugging aid.

4. If your program requires any user-defined execution-time files, issue the FILEDEF commands for them at this point. See the section of this book "User-defined Files" for more information.

5. Specify the default name MAIN that is assigned by the compiler unless you specified a name with the NAME option (see the section

Appendix A: FORTRAN Compilation Debug Facility

The FORTRAN Debug Facility statements (DEBUG, AT, DISPLAY, TRACE ON, and TRACE OFF) are described in the *IBM System/360 and System/370 FORTRAN IV Language*, Order No. GC28-6515. This section describes the output produced when these statements are used in a FORTRAN source module submitted to the Code and Go FORTRAN or FORTRAN IV (G1) compilers. These statements are not available if you specify the TEST option with either the FORTGI or GOFORT commands.

DEBUG Statement

The options UNIT, TRACE, SUBTRACE, INIT, and SUBCHK may be specified in the DEBUG statement. The UNIT option indicates the unit on which the DEBUG output is to be written. If the UNIT option is omitted, DEBUG output is written in the LISTING file.

TRACE

Trace output is written only when TRACE is on as a result of the TRACE ON statement. For each labeled statement that is executed, the line:

-DEBUG-TRACE *statement-label*

is written.

SUBTRACE

SUBTRACE is used to trace program flow from one routine to another. For each subprogram called, the line:

-DEBUG-SUBTRACE *subprogram-name*

is written on entry to the subprogram, and the line:

-DEBUG-SUBTRACE *RETURN*

is written on exit from the subprogram.

INIT

The output produced as a result of the INIT option is written regardless of any TRACE ON or TRACE OFF statements in the source module. Each time a value is assigned to an unsubscripted variable listed in the INIT option, the line:

-DEBUG- *variable-name = value*

is written, with the value given in the proper format for the variable type. When a value is assigned to an element of an array listed in the INIT option, the line:

-DEBUG- *array-name(element-number) = value*

is written, with the format of the value determined by the type of the array element. The single element number subscript is used regardless of the number of dimensions in the array.

SUBCHK

SUBCHK output is not affected by TRACE ON or TRACE OFF statements in the source module. When a reference to an array listed in the SUBCHK option includes subscripts such that the reference is outside the array, the line:

-DEBUG-SUBCHK *array-name(element-number)*

is printed. If the element number is negative, the number printed in the line is the arithmetic sum of 16,777,216 and the negative element number. For example, if the element number is -1, the number printed in the output line is 16,777,215. An attempt will be made, using the invalid subscripts, to execute the statement.

DISPLAY Statement

DISPLAY statement output is identical to NAMELIST WRITE output. The first line written is the name of the NAMELIST created by the compiler for the DISPLAY statement, preceded by the ampersand character:

& DBG *nn*#

where:

nn is the 2-digit decimal value assigned to the DISPLAY statement; this value begins at 01 for the first DISPLAY statement in the source module and increases by one for each subsequent DISPLAY statement.

The NAMELIST name is followed by the DISPLAY list, in NAMELIST FORMAT. The output is terminated with the line:

& END

Special Considerations

Any DEBUG output which is produced during an input/output operation is saved in storage until the input or output operation is complete. It is then written out. Saving this information may require additional storage space from the system. If the request cannot be satisfied, some of the DEBUG output may be lost. If this situation occurs, the message:

-DEBUG-SOME OUTPUT MISSING

is written after the output which was saved.

If a subscript appearing in an input/output list includes a function reference, and the FUNCTION contains a DISPLAY statement, the DISPLAY cannot be performed. In this case the message:

-DEBUG-DISPLAY DURING I/O SKIPPED

is written in the DEBUG output.

Appendix B: Assembler Language Subprograms

If you are an experienced FORTRAN programmer you can use assembler language subprograms with your FORTRAN main program. This section describes the linkage conventions that must be used by the assembler language subprogram to communicate with the FORTRAN main program. To understand this appendix, the reader must be familiar with the *Assembler Language* publication, Form GC28-6514 and the appropriate assembler language programmer's guide.

Subroutine References

You can refer to a subprogram in two ways: by a CALL statement or a function reference within an arithmetic expression.

Example:

```
CALL MYSUB( X,Y,Z )
I=J+K+MYFUNC( L,M,N )
```

For subprogram reference, the compiler generates:

1. A contiguous argument list; the addresses of the arguments are placed in this list to make the arguments accessible to the subprogram.

2. A save area in which the subprogram can save information related to the calling program.

3. A calling sequence to pass control to the subprogram.

Argument List

The argument list contains address of variables, arrays, and subprogram names used as arguments. Each entry in the argument list is four bytes and is aligned on a fullword boundary. The last three bytes of each entry contain the 24-bit address of an argument. The first byte of each entry contains zeros, unless it is the last entry in the argument list. If this is the last entry, the sign bit in the entry is set to 1.

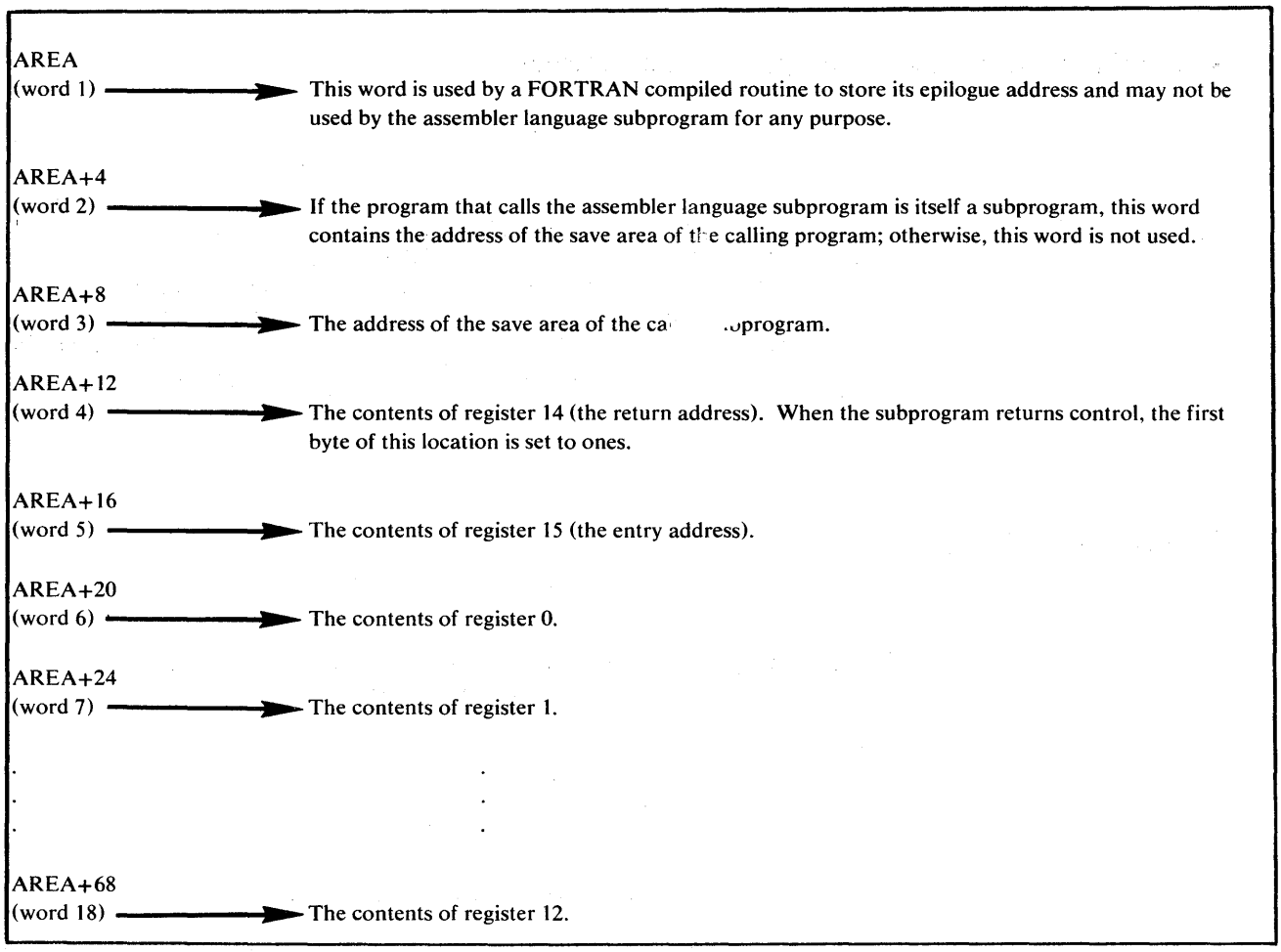


Figure 41. Save Area Layout and Word Contents

The address of the argument list is placed in general register 1 by the calling program.

Save Area

The calling program contains a save area in which the subprogram places information, such as the entry point for this program, an address to which the subprogram returns, general register contents, and addresses of save areas used by programs other than the subprogram. The amount of storage reserved by the calling program is 18 words. Figure 41 shows the layout of the save area and the contents of each word. The address of the save area is placed in general register 13.

The called subprogram does not have to save and restore floating-point registers.

Calling Sequence

A calling sequence is generated to transfer control to the subprogram. The address of the save area in the calling program is placed in general register 13. The address of the argument list is placed in general register 1, and the entry address is placed in general register 15. If there is no argument list, then general register 1 will contain zero. A branch is made to the address in register 15 and the return address is saved in general register 14. Figure 42 illustrates the use of the linkage registers.

Register Number	Register Name	Function
0	Result Register	Used for function subprograms only. The result is returned in general or floating-point register 0. However, if the result is a complex number, it is returned in floating-point registers 0 (real part) and 2 (imaginary part). *Note:* For subroutine subprograms, the result(s) is returned in a variable(s) passed by the programmer.
1	Argument List Register	Address of the argument list passed to the called subprogram.
2	Result Register	See Function of Register 0.
13	Save Area Register	Address of the area reserved by the calling program in which the contents of certain registers are stored by the called program.
14	Return Register	Address of the location in the calling program to which control is returned after execution of the called program.
15	Entry Point Register	Address of the entry point in the calling subprogram. *Note:* Register 15 is also used as a condition code register, a RETURN code register, and a STOP code register. The particular values that can be contained in the register are 16 - a terminal error was detected during execution of a subprogram (an IHCxxxI message is generated) 4*i- a RETURN i statement was executed n - a STOP n statement was executed 0 - a RETURN or a STOP statement was executed

Figure 42. Linkage Registers

Coding the Assembler Language Subprogram

Two types of assembler language subprograms are possible: the first type (lowest level) assembler subprogram does not call another subprogram; the second type (higher level) subprogram does call another subprogram.

Coding a Lowest Level Assembler Language Subprogram

For the lowest level assembler language subprogram, the linkage instructions must include:

1. An assembler instruction that names an entry point for the subprogram.
2. An instruction(s) to save any general registers used by the subprogram in the save area reserved by the calling program. (The contents of linkage registers 0 and 1 need not be saved.)
3. An instruction(s) to restore the "saved" registers before returning control to the calling program.
4. An instruction that sets the first byte in the fourth word of the save area to ones, indicating that control is returned to the calling program.
5. An instruction that returns control to the calling program.

Figure 43 shows the linkage conventions for an assembler language subprogram that does not call another subprogram. In addition to these conventions, the assembler program must provide a method to transfer arguments from the calling program and return the arguments to the calling program.

Name	Operation	Operand	Comments
deckname	start	0	branch around constants in calling
	bc	15,m+1+4(15)	sequence m must be an odd integer to insure
	dc	x'*m*'	that the program starts on a halfword
	dc	cl*m*' *name* '	boundary. The name can be padded with
*			blanks.
	stm	14,r,12(13)	the contents of registers 14, 15, and 0
*			through r are stored in the save area
*			of the calling program. r is any
*			number from 2 through 12.
	balr	b,0	establish base register (2 b 12)
	using	*,b	
		.	
		.	
		.	
		(user-written source statements)	
		.	
		.	
		.	
	lm	2,r,28(13)	restore registers
	mvi	12(13),x ff'	indicate control returned to calling program
	bcr	15,14	return to calling program

Figure 43. Linkage Conventions for Lowest Level Subprograms

Higher Level Assembler Language Subprogram

A higher level assembler subprogram must include the same linkage instructions as the lowest level subprogram, but because the higher level subprogram calls another subprogram, it must simulate a FORTRAN subprogram reference statement and include:

1. A save area and additional instructions to insert entries into its save area.

2. A calling sequence and a parameter list for the subprogram that the higher level subprogram calls.

3. An assembler instruction that indicates an external reference to the subprogram called by the higher level subprogram.

4. Additional instructions in the return routine to retrieve entries in the save area.

Note: If an assembler language main program calls a FORTRAN subprogram, the following instructions must be included in the assembler language program before the FORTRAN subprogram is called as follows:

```
L 15,=V( IBCOM# )
BAL 14,64( 15 )
```

These instructions cause initialization of return coding, interruption exceptions, and opening of the error message data set. If this is not done and the FORTRAN subprogram terminates either with a STOP statement or because of an execution-time error, the data sets opened by FORTRAN are not closed and the result of the termination cannot be predicted. Register 13 must contain the address of the save area that contains the registers to be restored upon termination of the FORTRAN subprogram. If control is to return to the assembler language subprogram, then register 13 contains the address of its save area. If control is to return to the operating system, then register 13 contains the address of its save area.

Figure 44 shows the linkage conventions for an assembler subprogram that calls another assembler subprogram.

Name	Operation	Operand	Comments
deckname	start	0	
	extrn	*name*$_2$	name of the subprogram called by this
	bc	15,*m*+1+4(15)	subprogram
	dc	x'*m*'	
	dc	cl*m*'*name*$_1$'	
*		save routine	
	stm	14,r,12(13)	the contents of register 14, 15, and 0
*			through r are stored in the save area of the
*			calling program. r is any number from 2
*			through 12.
	balr	b,0	establish base register
	using	*,b	
	lr	q,13	loads register 13, which points to the save
*			area of the calling program, into any
*			general register,q, except 0, 11, 13, and
*			15.
	la	13,area	loads the address of this program's save
*			area into register 13.
	st	13,8(0,q)	stores the address of this program's save
*			area into register 13.
	st	q,4(0,13)	stores the address of the previous save
*			area (the save area of the calling
*			program) into word 2 of this program's
*			save area
	bc	15,*prob*$_1$	
area	ds	18f	reserves 18 words for the save area
*		end of save routine	
prob$_1$		(*user-written program statements*)	
*		calling sequence	
	la	1,arglist	load address of argument list
	l	15,adcon	
	balr	14,15	
		(*more user-written program statements*)	
*		return routine	
	l	13,area+4	loads the address of the previous save
*			area back into register 13
	lm	2,r,28(13)	
	l	14,12(13)	loads the return address into register 14
	mvi	12(13),x'ff'	
	bcr	15,14	return to calling program
*			end of return routine
adcon	dc	a(*name*$_2$)	
*		argument list	
arglist	dc	al4(*arg*$_1$)	address of first argument
	.		
	.		
	.		
	dc	x'80'	indicate last argument in argument list
	dc	al3(*arg*$_n$)	address of last argument

Figure 44. Linkage Conventions for Higher Level Subprogram

In-Line Argument List

In coding your assembler program, you may establish an in-line argument list instead of an out-of-line list. In this case, you may substitute the calling sequence and argument list shown in Figure 42 for that shown in Figure 45.

```
adcon           dc          A( prob1 )
                .
                .
                .
                la          14,return
                l           15,adcon
                cnop        2,4
                balr        1,15
                dc          al4( arg1 )
                dc          al4( arg2 )
                .
                .
                .
                dc          x'80'
                dc          al3( argn )
return          bc          0,x' isn '
```

Figure 45. In-Line Argument List

Sharing Data in COMMON

Both named and blank COMMON in a FORTRAN IV program can be referred to by an assembler language subprogram. To refer to named COMMON, the V-type address constant is used.

Example:

name dc v(*name-of-COMMON)*

If a FORTRAN program has a blank COMMON area and blank COMMON is also defined (by the COM instruction) in an assembler language subprogram, *only* one blank COMMON area is generated for the output load module. Data in this blank COMMON is accessible to both programs.

To refer to blank COMMON, the following linkage may be specified:

```
        com
name    ds      0f
        .                                   .
        .                                   .
        .                                   .
-------------------------->     cname      csect
        l       11,=a( name )               .
        using name ,11                      .
                                            .
```

Retrieving Arguments From The Argument List

The argument list contains addresses for the arguments passed to a subprogram. The order of these addresses is the same as the order specified for the arguments in the calling statement in the main program. The address for the argument list is placed in register 1.

Example:

```
call mysub(a,b,c)
```

When this statement is compiled, the following argument list is generated.

00000000	address for A
00000000	address for B
10000000	address for C

For purposes of discussion, A is a real *8 variable, B is a subprogram name, and C is an array.

The address of a variable in the calling program is placed in the argument list. The following instructions in an assembler language subprogram can be used to move the real*8 variable A to location VAR in the subprogram.

```
l   q,0(1)
mvc var(8),0(q)
```

where

Q is any general register except 0.

For a subprogram reference, an address of a storage location is placed in the argument list. The address at this storage location is the entry point to the subprogram. The following instructions can be used to enter subprogram B from the subprogram to which B is passed as an argument.

```
l   q,4(1)
l   15,0(q)
balr14,15
```

where

Q is any general register except 0.

For an array, the address of the first variable in the array is placed in the argument list. An array [for example, a three-dimensional array C(3,2,2)] appears in this format in main storage.

C(1,1,1) C(2,1,1) C(3,1,1) C(1,2,1)
C(2,2,1) C(3,2,1) C(1,1,2) C(2,1,2)
C(3,1,2) C(1,2,2) C(2,2,2) C(3,2,2)

Figure 46 shows the general subscript format for arrays of 1, 2, and 3 dimensions.

Array A	Subscript Format
A(D1)	A(S1)
A(D1,D2)	A(S1,S2)
A(D1,D2,D3)	A(S1,S2,S3)
D1, D2, D3 are integer constants used in the DIMENSION statement. S1, S2, and S3 are subscripts used with subscripted variables.	

Figure 46. Dimension and Subscript Format

The address of the first variable in the array is placed in the argument list. To retrieve any other variables in the array, the displacement of the variable, that is, the distance of a variable from the first variable in the array, must be calculated. The formulas for computing the displacement (DISPLC) of a variable for one, two, and three dimensional arrays are

DISPLC=(S1-1)*L
DISPLC=(S1-1)*L+(S2-1)*D1*L
DISPLC=(S1-1)*L+(S2-1)*D1*L+(S3-1)*D2*D1*L

where:

L is the length of each variable in this array.

Example:

The variable C(2,1,2) in the main program is to be moved to a location ARVAR in the subprogram. Using the formula for displacement of integer variables in a three-dimensional array, the displacement (DISP) is calculated to be 28. The following instructions can be used to move the variable,

```
l   q,8(1)
l   r,disp
l   s,0(q,r)
st  s,arvar
```

where:

Q and R are any general register except 0.

S is any general register. Q and R cannot be general register 0.

Example: An assembler language subprogram is to be named ADDARR, and a real variable, an array, and an integer variable are to be passed as arguments to the subprogram. The statement

```
call addarr (X,Y,J)
```

is used to call the subprogram. Figure 47 shows the linkage used in the assembler subprogram.

```
Name    Operation   Operand
addarr  start       0
b       equ         8
        bc          15,12(15)
        dc          x'7'
        dc          cl7'addarr'
        stm         14,12,12(13)
        balr        b,0
        using       *,b
        l           2,8(1)              move 3rd argument to location called
        mvc         index(4),0(2)       index in assembler language subprogram.
        l           3,0(1)              move 1st argument to location called var
        mvc         var(4),0(3)         in assembler language subprogram.
        l           4,4(1)              load address of array into register 4.
                    ( user-written statements )
                      .
                      .
                      .
        lm          2,12,28(13)
        mvi         12(13),x'ff'
        bcr         15,14
        ds          0f
index   ds          1f
var     ds          1f
```

Figure 47. Assembler Subprogram Examples

Return I in an Assembler Language Subprogram

When a statement number is an argument in a CALL to an assembler language subprogram, the subprogram cannot access the statement number argument.

To accomplish the same thing as the FORTRAN statement RETURN *i* (used in FORTRAN subprograms to return to a point other than that immediately following the CALL), the assembler subprogram must place 4* *i* in register 15 before returning to the calling program.

Example:

When the statement

```
call sub (a,b,&10,&20)
```

is used to call an assembler language subprogram, the following instructions would cause the subprogram to return to the proper point in the calling program:

```
            .
            .
            .
la          15,4(toreturnto10)
bcr         15,14
            .
            .
            .
la          15,8(toreturnto20)
bcr         15,14
```

Object-Time Representation of FORTRAN Variables

The programmer who uses FORTRAN in connection with assembler language may need to know how the various FORTRAN data types appear in the computer. The following examples illustrate the object-time representation of FORTRAN variables as they appear under CMS.

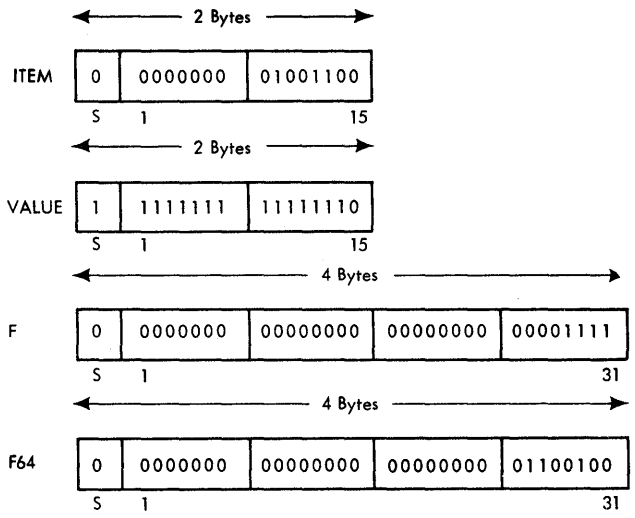
Integer Type

INTEGER variables are treated as fixed-point operands by all the compilers and are governed by the principles of System/370 fixed-point arithmetic. INTEGER variables are converted into either fullword (32 bit) or halfword (16 bit) signed integers.

Example:

```
integer*2 item/76/,value
integer*4 f,f64/100/
f = 15
value = -2
```

The value of the variables ITEM, VALUE, F, F64 appear in storage as follows:



where S in bit position 0 represents the sign bit. All negative numbers are represented in two's complement notation with a one in the sign-bit position.

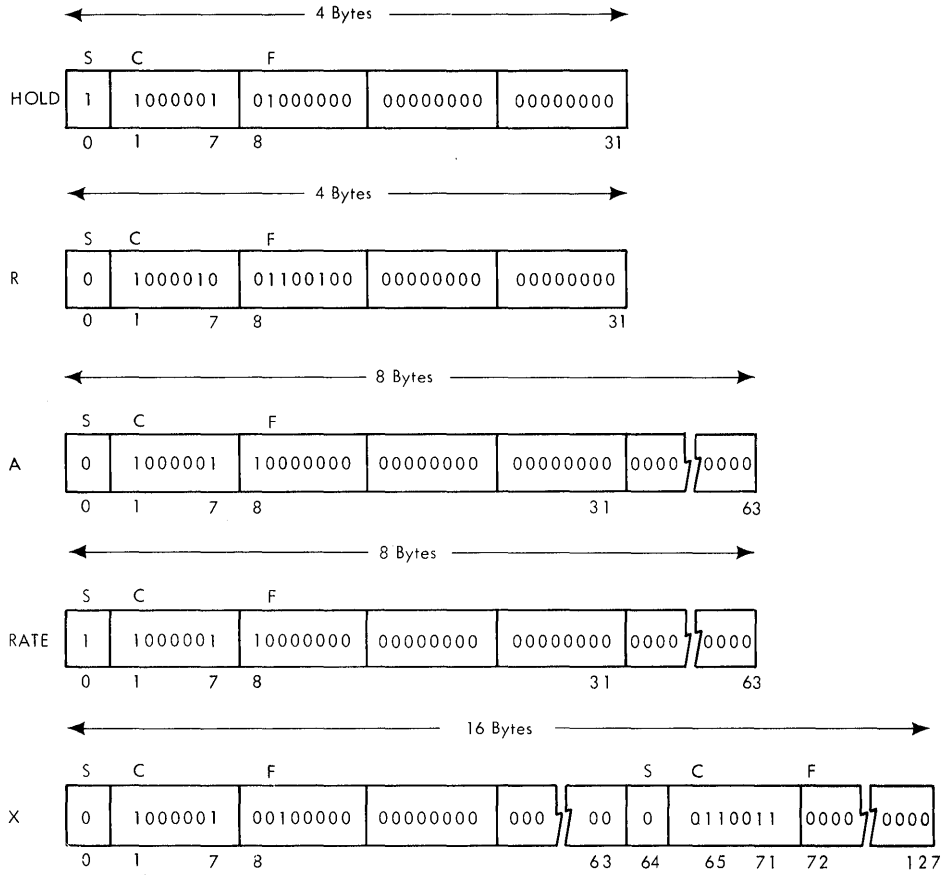
Real Type

All REAL variables are converted into short (32 bit) or long (64 bit) floating-point numbers by all the compilers. In addition, the H Extended compiler converts extended-precision REAL variable into extended (128 bit) floating-point numbers. The length of the numbers is determined by FORTRAN IV specification conventions.

Example:

```
real*4 hold,r/100./
real*8 a,rate/-8./
real*16 x
hold = -4.
a = 8.0d0
x = 2.0q0
```

The value of the variables HOLD, R, A, RATE, and X appear in storage as follows:



where:

s (sign bit) occupies bit position 0.

c (characteristic), or exponent, occupies bit positions 1 through 7.

f (fraction) occupies either bit positions 8 through 31 for a short, floating-point number, or bit positions 8 through 63 for long, floating-point number, or bit positions 8 through 63 and 72 through 127 for an extended-precision floating-point number (bit positions 64 through 71 represent a sign plus a characteristic having a value 14 less then the data represented in bits 0 through 7.

Note: Floating-point operations in System/360 may sometimes produce a negative zero, i.e., the sign bit of a floating-point zero will contain a one. FORTRAN IV compilers consider all floating-point numbers having a fraction of zero as equivalent. The setting of the sign bit is unpredictable in floating-point zeros computed by an object program. (A detailed explanation of floating-point operations can be found in the publication *IBM System/360: Principles of Operation,* Order No. GA22-6821.)

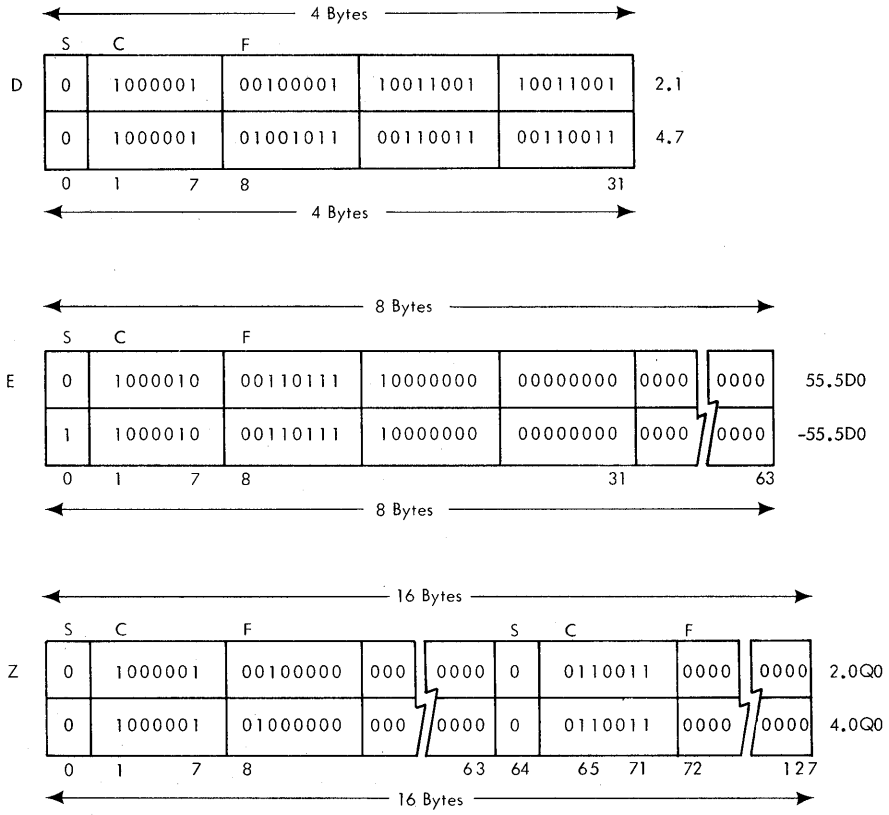
Complex Type

A COMPLEX variable has two parts (real and imaginary) and is treated as a pair of Real numbers. The COMPLEX parts are converted into two short, long, or extended floating-point numbers, depending upon the compiler.

Example:

```
complex d/(2.1,4.7)/,e*16,z*32
e=(55.5,-55.5)
z=(2.0Q0,4.0q0
```

The value of the variables D, E, and Z appear in storage as follows:



Logical Type

FORTRAN IV LOGICAL variables may specify only 2 values:

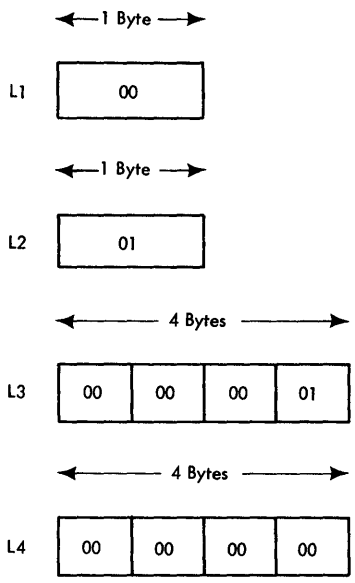
.TRUE. or .FALSE.

These logical values are assigned numerical values of '1' and '0', for .TRUE. and .FALSE., respectively.

Example:

```
logical*1 l1,l2/.true./
logical*4 l3,l4/.false./
l1 = .false.
l3 = .true.
```

The value of the variables L1, L2, L3, L4 to be assigned the following values (using hexadecimal notation):



Note: The values shown above for Logical variables are those assigned for the current implementation of the G1, Code and Go, and H Extended compilers. The assembler language programmer should not assume these values for future versions of these compiler, since they are subject to change.

The DUMP or PDUMP subroutine can also be used as an additional tool for understanding the object-time representation of FORTRAN data. Refer to the "Use of DUMP and PDUMP" publication *IBM System/360 Operating System: FORTRAN IV Library - Mathematical and Service Subprograms,* Order No. GC28-6818.

Appendix C: SIFT Utility

You can use the SIFT utility program to convert source programs written in free-form to fixed-form and vice versa. Since the Code and Go compiler can compile either fixed-form or free-form source programs, it is possible to convert free-form source programs written for Code and Go to fixed-form making them acceptable to the other FORTRAN IV compilers.

Converting Fixed-Form Input to Free-Form (Filetype of FORTRAN to Filetype of FREEFORT)

Fixed-form input to the SIFT utility program consists of fixed length 80-byte records. The last eight bytes of the fixed-form input record may optionally contain sequencing information. (The SIFT utility will ignore this sequencing information and place a unique sequence number in the first eight positions of the free-form line.)

When the SIFT utility is converting from fixed-form into free-form, it performs the following functions:

- Creates a variable-length, free-form record from each fixed-length, fixed-form record.
- Inserts a statement break character (-) at the end of each continued free-form line and deletes the continuation character from column 6 of the fixed-form continuation line.
- Changes any comment line by replacing the character C in column 1 with an asterisk (*).
- Creates a unique sequence number in columns 1 through 8 and places the statement number, text and break character in columns 9 through 81.

Converting Free-Form Input to Fixed-Form (Filetype of FREEFORT or FORTRAN to Filetype of FORTRAN)

Free-form input to the SIFT utility program consists of fixed-length, 80-byte records (filetype of FORTRAN) or variable-length records with a maximum length of 81 bytes (filetype of FREEFORT). The last eight bytes of fixed-length, free-form records may optionally contain sequencing information. If the input record is variable-length, the first eight bytes of the record must contain sequencing information. (The SIFT utility will ignore existing sequencing information and generate a new unique sequence number.)

When the SIFT utility is converting from free-form in to fixed-form, it performs the following functions:

- Creates one or more fixed-length, 80-byte records from each free-form line.
- Deletes the statement break character (-) at the end of each continued free-form line and inserts a continuation character in column 6 of each fixed-form continuation line.
- Changes any free-form comment line by replacing the asterisk (*) or double quote (") with the character C in column 1.
- Begins any statement label in column 1 and begins the text of the FORTRAN statement in column 7 of the output record.
- Creates a unique sequence number in columns 73 through 80 of the fixed-form record.
- Combines free-form continuation lines, if possible.

Invoking the SIFT Utility

The SIFT utility is invoked by the CONVERT command. The format of the command follows:

CONVERT *filename1 filename2* GOFORT ([FIXED | FREE] [NOLIST])

where:

CONVERT — is a required part of the command and must always appear.

filename1 — specifies the filename of the file to be converted. If you want to convert a free-form file to fixed-form, *filename1* must have a filetype of either FREEFORT or FORTRAN, and you must specify the FIXED option. (If both a FREEFORT and FORTRAN file exist, then the FREEFORT file will be converted.) If you want to convert a fixed-form file to free-form, *filename1* must have a filetype of FORTRAN, and you must specify the FREE option.

filename2 — specifies a unique filename (that is, one that does not already exist). You must supply this name, which will be assigned to the output file that will contain the converted records. If the file created for *filename2* has the same file identifier as a file that already exists, the existing file will be replaced by the new file. If *filename1* has a filetype of FREEFORT or FORTRAN, contains free-form records, and the FIXED option is specified, *filename2* will be created with a filetype of FORTRAN and will contain fixed-form records. If *filename1* has a filetype of FORTRAN, contains fixed-form records, and

the FREE option is specified, *filename2* will be created with a filetype of FREEFORT and will contain variable-length, free-form records.

GOFORT — is a required part of the command and must always be typed.

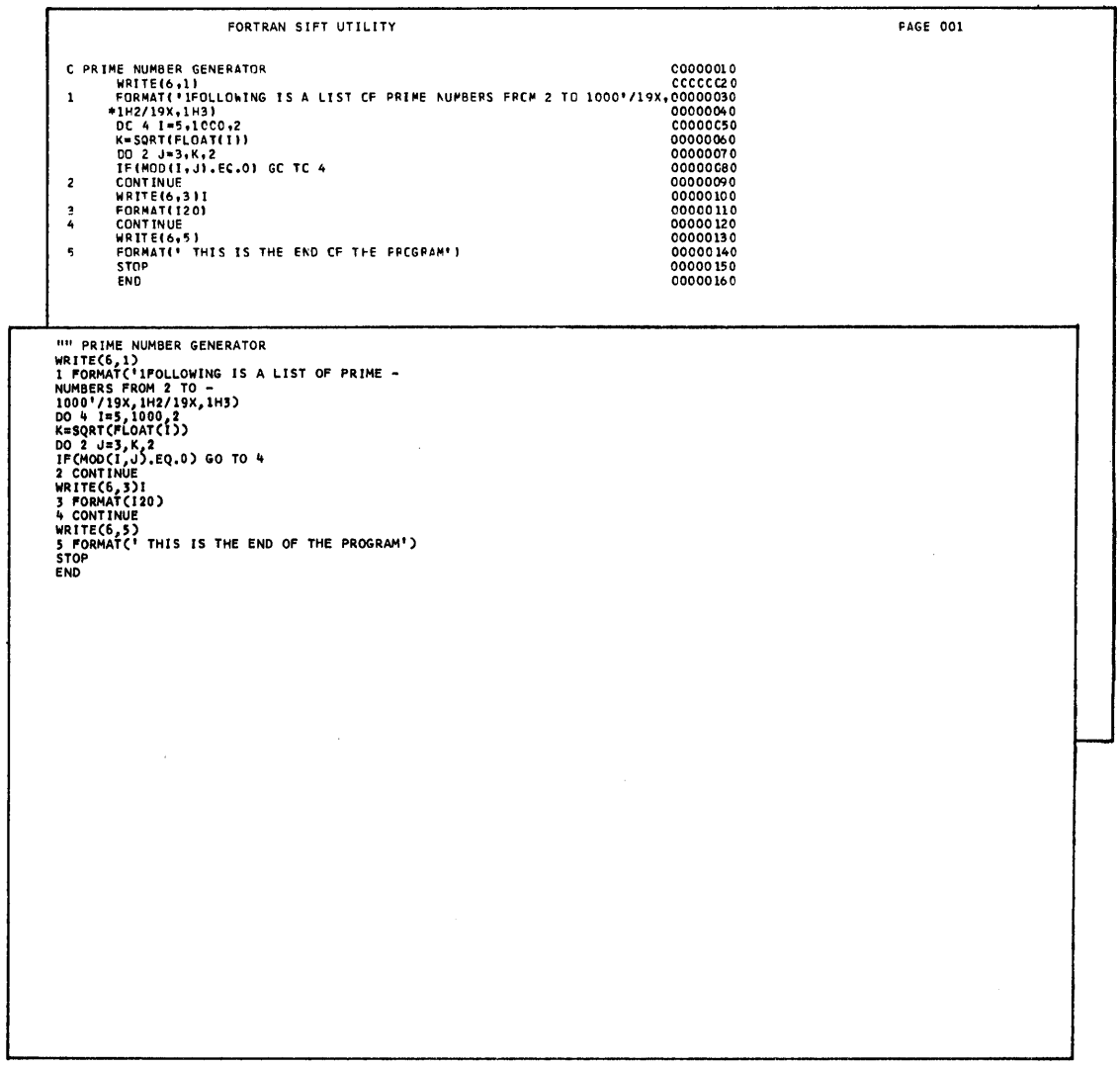
FIXED — indicates that you want to convert a file with a filetype of FREEFORT or FORTRAN containing free-form records to a file with a filetype of FORTRAN containing fixed-form records. This is the default if no option is specified.

FREE — indicates that you want to convert a file with a filetype of FORTRAN containing fixed-form records to a file with a filetype of FREEFORT containing variable-length free-form records. If you omit this option, FIXED is assumed.

NOLIST — is optional; however, if specified, the listing of the converted program that is normally typed at your terminal is not produced. If specified, this option must be placed last in the command.

Note: To convert fixed-length, free-form records to variable-length, free-form records, you have to use the CONVERT command twice. First, you must convert from free-form to fixed-form, and then you convert the newly produced fixed-form records back into free-form. The free-form source thus created will consist of variable length records.

Figure 48 shows a sample free form source program, and the resultant fixed form program that is created from it by the SIFT utility.



```
                    FORTRAN SIFT UTILITY                                                    PAGE 001


C PRIME NUMBER GENERATOR                                                          C0000010
      WRITE(6,1)                                                                  CCCCCC20
1     FORMAT('1FOLLOWING IS A LIST CF PRIME NUMBERS FRCM 2 TO 1000'/19X,          00000030
     *1H2/19X,1H3)                                                                00000040
      DC 4 I=5,1CC0,2                                                             C0000C50
      K=SQRT(FLOAT(I))                                                            00000060
      DO 2 J=3,K,2                                                                00000070
      IF(MOD(I,J).EC.0) GC TC 4                                                   00000080
2     CONTINUE                                                                    00000090
      WRITE(6,3)I                                                                 00000100
3     FORMAT(I20)                                                                 00000110
4     CONTINUE                                                                    00000120
      WRITE(6,5)                                                                  00000130
5     FORMAT(' THIS IS THE END CF THE PRCGRAM')                                   00000140
      STOP                                                                        00000150
      END                                                                         00000160
```

```
"" PRIME NUMBER GENERATOR
WRITE(6,1)
1 FORMAT('1FOLLOWING IS A LIST OF PRIME -
NUMBERS FROM 2 TO -
1000'/19X,1H2/19X,1H3)
DO 4 I=5,1000,2
K=SQRT(FLOAT(I))
DO 2 J=3,K,2
IF(MOD(I,J).EQ.0) GO TO 4
2 CONTINUE
WRITE(6,3)I
3 FORMAT(I20)
4 CONTINUE
WRITE(6,5)
5 FORMAT(' THIS IS THE END OF THE PROGRAM')
STOP
END
```

Figure 48. Free-Form Fixed-Form SIFT Output Listing

Appendix D. Subprograms for the Extended Error Handling Facility

The following information is for the use of systems programmers who need to make temporary changes to the extended error handling option table. As such changes are not the concern of most programmers, no attempt is made to explain the material on an elementary level. If you do not need to modify the option table, skip this appendix entirely.

IBM provides four subroutines for use in extended error handling: ERRSAV, ERRSTR, ERRSET, and ERRTRA. These subroutines allow access to the option table to alter it dynamically. (Certain option table entries are protected against alteration when the option table is set up. If a request is made by means of CALL ERRSTR or CALL ERRSET to alter such an entry, the request is ignored. See Figure 15 to determine which IBM-supplied option table entries cannot be altered.) Changes made dynamically are in effect for the duration of the session in which the change was made. Only the current copy of the option table in main storage is affected; the copy in the FORTRAN library remains unchanged. All passed parameters, unless otherwise indicated, are 4-byte (fullword) integers.

Accessing and Altering the Option Table Dynamically

1. The CALL ERRSAV statement, described below, can be used for temporarily modifying an entry. This statement causes an option table entry to be copied into an 8-byte storage area accessible to the FORTRAN programmer. The format is:

   CALL ERRSAV *(ierno,tabent)*

   where:

   ierno

   is the error number to be referenced in the option table. Should any number not within the range of the option table be used, an error message will be printed.

   tabent

   is the name of an 8-byte storage area where the option table entry is to be stored.

   Example:

```
call errsav(215,alterx)
```

2. To store an entry in the option table, the following statement is used:

CALL ERRSTR *(ierno,tabent)*

where:

ierno

is the error number for which the entry is to be stored in the option table. Should any number not within the range of the option table be used, an error message will be printed.

tabent

is the name of an 8-byte storage area containing the table entry data.

Example:

```
call errstr(215,alterx)
```

3 The CALL ERRSET statement permits the user to change up to five different options in an option table entry. It consists of six parameters. The last parameters are optional, but each omitted parameter must have its place noted by a comma and a zero if succeeding parameters are specified. (Omitted parameters at the end of the list require no place notation.) CALL ERRSET has the format:

CALL ERRSET *(ierno,inoal,inomes,itrace,iusadr,irange)*

where:

ierno

is the error number to be referenced in the option table. Should any number not within the range of the option table be used, an error message will be printed. (Note that if *ierno* is specified as 212, there is a special relationship between the ierno and *irange* parameters. See the explanation for *irange.)*

inoal

is an integer specifying the number of errors permitted before execution is terminated. If *inoal* is specified as either zero or a negative number, the specification is ignored, and the number-of-errors option is not altered. If a value of more than 255 is specified, an unlimited number of errors is permitted.

inomes

is an integer indicating the number of messages to be printed. A negative value specified for inomes causes all messages to be suppressed; a specification of 0 indicates that the number-of-messages option is not to be altered.

itrace

is an integer whose value may be 0, 1, or 2. A specification of 0 indicates the option is not to be changed; a specification of 1 requests that no traceback be printed after an error occurrence; a

specification of 2 requests the printing of a traceback after each error occurrence. (If a value other that 1 or 2 is specified, the option remains unchanged.)

iusadr specifies one of the following:

a. the value 1, as a 4-byte integer, indicating that the option table is to be set to show no user-exit routine (that is, standard corrective action is to be used when continuing execution).

b. the name of a closed subroutine that is to be executed after the occurrence of the error identified by *ierno* . The name must appear in an EXTERNAL statement in the source program, and the routine to which, control is to be passed must be made available via a GLOBAL TXTLIB command, or in the source program itself.

c. The value 0, indicating that the table entry is not to be altered.

range

serves a double function. It specifies one of the following:

a. An error number higher than that specified in *ierno* . The number indicates that the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by *ierno* and *range* . (If *irange* specifies a number lower than *ierno,* the parameter is ignored, unless *ierno* specifies the number 212.)

b. A print control character if *ierno* specified 212. The value 1 is specified to provide single spacing for an overflow line (standard fixup for WRITE statements). If a value other than 1 is specified, no print control is provided.

The default value 0 is assumed if the parameter is omitted (i.e., no print control is provided, and the values specified for all parameters apply only to the error condition in *ierno).*

Examples:

```
call errset (310, 20,5, 0, myerr, 320)

call errset (212, 10, 5, 2, 1, 1)

call errset (212, 0, 0, 0, 0, 1)
```

The first example specifies the following:

a. error condition 310 *(ierno)*

b. the error condition may occur up to 20 times *(inoal)*

c. the corresponding error message may be printed up to 5 times) *(inomes)*

d. the default for traceback information is to remain in effect *(itrace)*

e. the user-written routine MYERR is to be executed after each error occurrence *(insadr)*

f. the same options are to apply to all error conditions from 310 to 320 *(irange)*

The second example specifies:

a. error condition 212

b. the condition may occur up to 10 times

c. the corresponding message may be printed up to 5 times

d. traceback information is to be displayed after each error occurrence

e. standard corrective action is to be executed after an error

f. print control is to be employed

For purposes of illustration, this example explicitly specifies all default options except in requesting print control.

The third example illustrates an alternative method of specifying exactly the same options as the second example. It states that no changes are to be made to default settings except in requesting print control.

4 The CALL ERRTRA statement permits the user to dynamically request a traceback and continued execution. It has the format:

```
CALL ERRTRA
```

The call has no parameters.

User-Supplied Error Handling

The user has the ability of calling, in his own program, the FORTRAN error monitor (ERRMON) routine, the same routine used by FORTRAN itself when it detects an error. ERRMON examines the option table for the appropriate error number and its associated entry and takes the actions specified. If a user-exit address has been specified, ERRMON transfers control to the user-written routine indicated by that address. Thus, the user has the option of handling errors in one of two ways: (1) simply by calling ERRMON -- without supplying a user-written exit routine; or (2) by calling ERRMON and providing a user-written exit routine.

In either case, certain planning is required at the installation level. For example, error numbers must be assigned to error conditions to be detected

by the user, and additional option table entries must be made available for these conditions. The routine that uses the error monitor for error service should have the status of an installation general-purpose function similar to the IBM-supplied mathematical functions. The number of installation error conditions must be known when the FORTRAN library is created at program installation, so that entries will be provided in the option table. The error numbers chosen for user subprograms are restricted in range. IBM-designated error conditions have reserved error codes from 000 to 301. Error codes for installation-designated error situations must be assigned in the range 302 to 899. The error code is used by FORTRAN to find the proper entry in the option table.

To call the ERRMON routine, the following statement is used:

CALL ERRMON *(imes,iretcd,ierno,data1,data2,...)*

where:

imes

is the name of an array aligned on a fullword boundary, which contains, in EBCDIC characters, the text of the message to be printed. The number of the error condition should be included as part of the text, because the error monitor prints only the text passed to it. The first item of the array contains an integer whose value is the length of the message. Thus, the first four bytes of the array will not be printed. If the message length is greater than the length of the buffer, it will be printed on two or more lines of printed output.

iretcd

is an integer variable made available to the error monitor for the setting of a return code. The following codes can be set:

0 - The option table or user-exit routine indicates that standard correction is required.

1 - The option table indicates that a user exit to a corrective routine has been executed. The function is to be re-evaluated using arguments supplied in the parameters *data1,data2....* For input/output type errors, the value 1 indicates that standard correction is not wanted.

ierno

is the error condition number in the option table. Should any number not within the range of the option table be specified, an error message will be printed.

data1,data2,...

are variable names in an error-dectecting routine for the passing of arguments found to be in error. One variable must be specified for each argument. Upon return to the error-detecting routine, results obtained from corrective action are in these variables. Because the content of the variables can be altered, the locations in which they are placed should be used only in the CALL statement to the error monitor; otherwise, the user of the function may have literals or variables destroyed.

Because *data1* and *data2* are the parameters which the error monitor will pass to a user-written routine to correct the detected error, care must be taken to make sure that these parameters agree in type and number in the call to ERRMON and in a user-written corrective routine, if one exists.

Example:

```
call errmon(mymsg,icode, 315,d1,d2)
```

The example states that the message to be printed is contained in an array named MYMSG, the field named ICODE is to contain the return code, the error condition number to be investigated is 315, and arguments to be passed to the user-written routine are contained in fields named D1 and D2.

Figure 49 illustrates the use of the CALL ERRSET and CALL ERRMON statements in a program using a user-supplied subprogram to handle divide-by-zero conditions.

```
c       main program that uses the subroutine divide
        common e
        external fixdiv
c       set up option table with address of user exit
C
        call errset(302,30,5,1,fixdiv)
c
        e=0
c       get values to call divide with
        read(5,9) a,b
        if(b) 1,2,1
2       e =1.0
1       call divide(a,b,c)
        write(6,10)c
9       format(2e20.8)
10      format('1',e20.8)
        stop
        end
        subroutine divide(a,b,c)
c       routine to perform the calculation c=a/b
c       if b=0 then use error message facility to service error
c       provide message to be printed
        dimension mes(4)
        data mes(1)/12/,mes(2)/' dov'/,mes(3)'302i/,mes(4)/' b=0'/
        data rmax/z7fffffff/
c       message to be printed is
c       div302i b=0
c       error code 302 is available and assigned to this routine
c       step1 save a,b in local storage
        d=a
        e=b
c       step2 test for error condition
100     if(e) 1,2,1
c       normal case -- compute function
1       c=d/e
        return
c       step3 error detected call error monitor
c
2       call errmon(mes,iretcd,302,d,e)
c
c       step4 be ready to accept a return from the error monitor
        if(iretcd) 5,6,5
c       if iretcd=0 standard result is desired
c       standard result will be c=largest number if d is not zero
c       cr c=0 if e=0 and d=0
6       if(d) 7,8,7
c       c=0.0
        go to 9
7       c=rmax
9       return
c       user fix up indicated.  recompute with new value placed in e
5       go to 100
        end
        subroutine fixdiv(iretcd,ino,a,b)
```

Figure 49. Sample Program Using Extended Error Handling Facility (Part 1 of 2)

```
c       this is a user exit to serve the subroutine divide
c       the parameters in the call match those used in the call to
c       errmon made by subroutine divide
c       step1 is alternate value for b available -- main program
c       has supplied a new value in e. if e=0 no new value is available
        common e
        if(e) 1,2,1
c       new value available take user correction exit
1       b=e
        return
c       new value not available use standard fix up
2       iretcd=0
        return
        end
```

Figure 49. Sample Program Using Extended Error Handling Facility (Part 2 of 2)

User-supplied Exit Routine

When a user-exit address is supplied in the option table entry for a given error number, the error monitor calls the specified subroutine for corrective action. The subroutine may be user-written and is called by the assembler language code equivalent to the following statement:

CALL *x (iretcd,ierno,data1,data2...)*

where:

x is the name of the routine whose address was placed into the option table by the *iusadr* parameter of the CALL ERRSET statement. (Interpretation of the other parameters -- *iretcd,ierno,data1,data2* -- are the same as those for the CALL ERRMON statement.) If an input/output error is detected (that is, an error for codes 211 to 237), subroutine *x* must not execute any FORTRAN I/O statements, that is, READ, WRITE, BACKSPACE, END FILE, REWIND, PAUSE, or any calls to PDUMP or ERRTRA. Similarly, if errors for codes 216 or 241-301 occur, the subroutine *x* must not call the library routine that detected the error or any routine which uses that library routine. For example, a statement such as

```
r = a**b
```

cannot be used in the exit routine for error 252, because the FORTRAN library subroutine FRXPR# uses EXP, which detects error 252.

Note that although a user-written corrective routine may change the setting of the return code *(iretcd),* such a change is subject to the following restrictions:

1. If *iretcd* is set to 0, then *data1* and *data2* must not be altered by the corrective routine, since standard corrective action is requested. If *data1* and *data2* are altered when *iretcd* is set to 0, the operations that follow will have unpredictable results.

2. Only the values 0 and 1 are valid for *iretcd*. A user-exit routine must ensure that one of these values is used if it changes the return code setting. Note too, that the user-written exit routine can be written in FORTRAN or in assembler language. In either case, it must be able to accept the call to it as shown above. The user-exit routine must be a closed subroutine that returns control to the caller.

   If the user-written exit routine is written in assembler language, the end of the parameter list can be checked. The high-order byte of the last parameter will have the hexadecimal value 80. If the routine is written in FORTRAN, the parameter list must match in length the parameter list passed in the CALL statement issued to the error monitor.

When the extended error handling facility encounters a condition or a request that requires user notification, an informational message is printed.

The error monitor is not recursive: If it has laready been called for an error, it cannot be re-entered if the user-written corrective routine cuases any of the error conditions that are listed in the option table.

Actions the user may take if he wishes to correct an error are described in Figures 50, 51, and 52.

Error Code	Parameters Passed to User	Standard Corrective Action	User-Supplied Corrective Action
206	A,B,I	I = low order part of number for input too large.	User may alter I (see note 3).
211	A,B,C	Treat format field containing C as end of FORMAT statement.	(a) If compiled FORMAT statement, put hexadecimal character in C (see note 1). (b) If variable format, move EBCDIC character into C (see note 1).
212	A,B,D	*Input:* Ignore remainder of I/O list. *Output:* Continue by starting new output record. Supply carriage control character if required by option table.	See note 2.
213	A,B,D	Ignore remainder of I/O list.	See note 2.
214	A,B,D	*Input:* Ignore remainder of I/O list; ignore I/O request for ASCU tape. *Output:* If unformatted write initially requested, change record format to vs. If formatted write initially requested, ignore I/O request.	If user correction is requested, the remainder of the I/O list is ignored.
215	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character. I/O operations may not be performed (see note 1).
217	A,B,D	Increment FORTRAN sequence number and read next file.	See note 2.
218[4]	A,B,D,F	Ignore remainder of I/O list.	See note 2.
219[5]-224	A,B,D	Ignore remainder of I/O list.	See note 2.
225	A,B,E	Substitute zero for the invalid character.	The character placed in E will be substituted for the invalid character (see note 1).
226	A,B,R	R = 0 for input number too small. R = $10^{63} - 1$ for input number too large.	User may alter R (see note 3).
227	A,B,D	Ignore remainder of I/O list.	See note 2.
228	A,B,D	Ignore remainder of I/O list.	See note 2.
229	A,B,D	Move 256 characters and resume processing with the next constant beyond the count given.	See note 2.
231	A,B,D	Ignore remainder of I/O list.	See note 2.
232	A,B,D,G	Ignore remainder of I/O list.	See note 2.
233	A,B,D	Change record number to list maximum allowed (32,000).	See note 2.
234-236	A,B,D	Ignore remainder of I/O list.	See note 2.
237	A,B,D,F	Ignore remainder of I/O list.	See note 2.
238	A,B,D	Ignore remainder of I/O list.	See note 2.
239	A,B,D	Ignore remainder of I/O list.	See note 2.

MEANINGS:

A–Address of return code field (INTEGER*4)
B–Address of error number (INTEGER*4)
C–Address of invalid format character (LOGICAL*1)
D–Address of data set reference number (INTEGER*4)
E–Address of invalid character (LOGICAL*1)
F–Address of DECB
G–Address of record number requested (INTEGER*4)
I–Result after conversion (INTEGER*4)
R–Result after conversion (REAL*4)

NOTES:

1. Alternatively, the user can set the return code to 0, thus requesting a standard corrective action.
2. The user can do anything he wishes except perform another I/O operation – i.e., issue a READ, WRITE, BACKSPACE, END FILE, REWIND, PAUSE, PDUMP, or ERRTRA. On return to the library, the remainder of the I/O request will be ignored.
3. The user exit routine may supply an alternative answer for the setting of the result register. The routine should always set an INTEGER*1, variable and the FORTRAN library will load fullword or halfword depending on the length of the argument causing the error.
4. If error condition 218 (I/O error detected) occurs while error messages are being written on the object error data set, the message is written on the console sheet and the job is terminated.
5. If no FILEDEF command has been supplied for the object error data set, error message IHN219I or IHO219I is written on the console sheet and the job is terminated.

Figure 50. Corrective Action After Error Occurrence

Error Code	FORTRAN Reference	Invalid Argument Range	Options	
			Standard Corrective Action	User Supplied Corrective Action (See Note 1)
216	CALL SLITE (I)	I>4	The call is treated as a no operation	I
216	CALL SLITET (I,J)	I > 4	J=2	I
241	K=I**J	I=0, J≤0	K=0	I, J
242	Y=X**I	X=0, I≤0	If I=0, Y=1 If I<0, Y=*	X, I
243	DA=D**I	D=0, I≤0	If I=0, Y=1 If I<0, Y=*	D, I
244	XA=X**Y	X=0, Y≤0	XA=0	X, Y
245	DA=D**DB	D=0, DB≤0	DA=0	D, DB
246	CA=C**I	C=0+0i, I≤0	If I=0, C=1+0i If I<0, C=*+0i	C, I
247	CDA=CD*I	C=0+0i, I≤0	If I=0, C=1+0i If I<0, C=*+0i	CD, I
251	Y=SQRT (X)	X<0	$Y=\|X\|^{1/2}$	X
252	Y=EXP (X)	X>174.673	Y=*	X
253	Y=ALOG (X)	X=0 X<0	Y=* Y=log|X|	X X
	Y=ALOG10 (X)	X=0 X<0	Y=-* $Y=\log_{10}\|x\|$	X X
254	Y=COS (X) Y=SIN (X)	$\|X\| \geq 2^{18} * \pi$	$Y=\frac{\sqrt{2}}{2}$	X
255	Y=ATAN2 (X, XA)	X=0, XA=0	Y=0	X, XA
256	Y=SINH (X) Y=COSH (X)	$\|X\| \leq 175.366$	Y=(sign x)* Y=*	X

Variable	*Type*
I, J	Variables of INTEGER*4
X, XA, Y	Variables of REAL*4
D, DA, DB	Variables of REAL*8
C, CA	Variables of COMPLEX*8
Z, X_1, X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	Variables of COMPLEX*16

Notes:
1. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.
2. The largest number that can be represented in floating point is indicated above by *.
3. The value e=approximately 2.7183.

Figure 51. Corrective Action After Mathematical Subroutine Error Occurrence (Part 1 of 4)

Error Code	FORTRAN Reference	Invalid Argument Range	Options: Standard Corrective Action	Options: User-Supplied Corrective Action (See Note 1)
257	Y=ARSIN (X) Y=ARCOS (X)	$\|X\|>1$	If $X>1.0$, ARCSIN $(X)=\frac{\pi}{2}$ If $X<-1.0$, ARCSIN $(X)=-\frac{\pi}{2}$	X
258	Y=TAN (X) Y=COTAN (X)	$\|X\|\geq(2^{18})*\pi$	If $X>1.0$, ARCOS $=0$ If $X<-1.0$, ARCOS $=\pi$	X
259	Y=TAN (X)	X is too close to an odd multiple of $\frac{\pi}{2}$	Y=*	X
260	Y=COTAN (X)	X is too close to a multiple of π	Y=*	X
261	DA=DSQRT (D)	$D<0$	$DA=\|D\|^{1/2}$	D
262	DA=DEXP (D)	$D>176.673$	DA=*	D
263	DA=DLOG (D) DA=DLOG10 (D)	$D=0$ $D<0$ $D=0$ $D<0$	DA=-* DA=log|X| DA=-* DA=$\log_{10}\|X\|$	D D D
264	DA=DSIN (D) DA=DCOS (D)	$\|D\|\geq 2^{50}*\pi$	DA=$\frac{\sqrt{2}}{2}$	D
265	DA=DATAN2(D,DB)	D=0, DB=0	DA=0	D, DB
266	DA=DSINH (D) DA=DCOSH (D)	$\|D\|\geq 175.366$	DA=(sign X)* DA=*	D

Variable	*Type*
I, J	Variables of INTEGER*4
X, XA, Y	Variables of REAL*4
D, DA, DB	Variables of REAL*8
C, CA	Variables of COMPLEX*8
Z, X_1, X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	Variables of COMPLEX*16

Notes:
1. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.
2. The largest number that can be represented in floating point is indicated above by *.
3. The value e=approximately 2.7183.

Figure 51. Corrective Action After Mathematical Subroutine Error Occurrence (Part 2 of 4)

Error Code	FORTRAN Reference	Invalid Argument Range	Options: Standard Corrective Action	Options: User-Supplied Corrective Action (See Note 1)
267	DA=DARSIN (D)	$\|D\|>1$	If $X>1.0$, DARSIN $(X)=\frac{\pi}{2}$ If $X<-1.0$, DARSIN $(X)=-\frac{\pi}{2}$	D
	DA=DARCOS (D)		If $X>1.0$, DARCOS=0 If $X<-1.0$, DARCOS=π	
268	DA=DTAN (D) DA=DCOTAN (D)	$\|X\|\geq 2^{50}*\pi$	DA=1	D
269	DA=DTAN (D)	D is too close to an odd multiple of 2π	DA=*	
	DA=DCOTAN (D)	D is too close to a multiple of π	DA=*	D
For errors 271 through 275, $C=X_1+iX_2$				
271	Z=CEXP (C)	$X_1>174.673$	$Z=*(\cos X_2+\sin X_2)$	C
272	Z=CEXP (C)	$\|X_2\|\geq 2^{18}*\pi$	$Z=e^{X1}+0*i$	C
273	Z=CLOG (C)	C=0+0i	z=-*+0i	C
274	Z=CSIN (C) Z=CCOS (C)	$\|X_1\|\leq 2^{18}*\pi$	$Z=0+\sinh(X_2)*i$ $Z=\cosh(X_2)+0*i$	C
275	Z=CSIN (C)	$X_2>174.673$	$Z=\frac{*}{2}(\sin X_1+i\cos X_1)$	C
	Z=CCOS (C)		$Z=\frac{*}{2}(\cos X_1-i\sin X_1)$	C
	Z=CSIN (C)	$X_2<-174.673$	$Z=\frac{*}{2}(\sin X_1-i\cos X_1)$	C
	Z=CCOS (C)		$Z=\frac{*}{2}(\cos X_1+i\sin X_1)$	C
For errors 281 through 285, $CD=X_1+iX_2$				
281	Z=CDEXP (CD)	$X_1>174.673$	$Z=*(\cos X_2+i\sin X_2)$	CD

Variable	*Type*
I, J	Variables of INTEGER*4
X, XA, Y	Variables of REAL*4
D, DA, DB	Variables of REAL*8
C, CA	Variables of COMPLEX*8
Z, X_1, X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	Variables of COMPLEX*16

Notes:
1. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.
2. The largest number that can be represented in floating point is indicated above by *.
3. The value e=approximately 2.7183.

Figure 51. Corrective Action After Mathematical Subroutine Error Occurrence (Part 3 of 4)

Error Code	FORTRAN Reference	Invalid Argument Range	Options	
			Standard Corrective Action	User-Supplied Corrective Action (See Note 1)
282	Z=CDEXP (CD)	$\|X_2\| \geq 2^{50} * \pi$	$Z = e^{1} + 0 * i$	CD
283	Z=CDLOG (CD)	CD=0+0i	Z=-*+0i	CD
284	Z=CDSIN (CD) Z=CDCOS (CD)	$\|X_1\| \geq 2^{50} * \pi$	Z=0+SINH (X_2)*i Z=COSH (X_2)+0*i	CD
285	Z=CDSIN (CD)	$X_2 > 174.673$	$Z = \frac{*}{2}(\text{SIN } X_1 + i\text{COS } X_1)$	CD
	Z=CDCOS (CD)		$Z = \frac{*}{2}(\text{COS } X_1 - i\text{SIN } X_1$	CD
	Z=CDSIN (CD)	$X_2 < -174.673$	$Z = \frac{*}{2}(\text{SIN } X_1 - i\text{COS } X_1)$	CD
	Z=CDCOS (CD)		$Z = \frac{*}{2}(\text{COS } X_1 + i\text{SIN } X_1)$	CD
290	Y=GAMMA (X)	$X \leq 2^{-252}$ or $X \geq 57.5744$	Y=*	X
291	Y=ALGAMA (X)	$X \leq 0$ or $X \geq 4.2937 * 10^{73}$	Y=*	X
300	DA=DGAMMA (D)	$D \leq 2^{-252}$ or $D \geq 57.5774$	DA=*	D
301	DA=DLGAMA (D)	$D \leq 0$ or $D \geq 4.2937 * 10^{73}$	DA=*	D

Variable	*Type*
I, J	Variables of INTEGER*4
X, XA, Y	Variables of REAL*4
D, DA, DB	Variables of REAL*8
C, CA	Variables of COMPLEX*8
Z, X_1, X_2	Complex variables to be given the length of the functioned argument when they appear.
CD	Variables of COMPLEX*16

Notes:
1. The user-supplied answer is obtained by recomputation of the function using the value set by the user routine for the parameters listed.
2. The largest number that can be represented in floating point is indicated above by *.
3. The value e=approximately 2.7183.

Figure 51. Corrective Action After Mathematical Subroutine Error Occurrence (Part 4 of 4)

Program Interrupt Messages			Options	
Error Code	**Parameters Passed to User Exit**	**Reason for Interrupt[1]**	**Standard Corrective Action**	**User-Supplied Corrective Action**
207	D,I	Exponent Overflow (Interrupt Code 12).	Result register set to the largest possible floating point number. The sign of the result register is not altered.	User may alter D.[2]
208	D,I	Exponent underflow (Interrupt Code 13)	The result register is set to zero.	User may alter D.[2]
209	None	Divide check: Integer divide (Interrupt code 9), Decimal divide (Interrupt Code 11), Floating point divide (Interrupt Code 15).[3]	None.	See Note 4.

Variable	*Type*	*Description*
D	A variable REAL*8	This variable contains the contents of the result register after the interrupt.
I	A variable INTEGER*4	The variable contains the "exponent" as an integer value for the number in D. It may be used to determine the amount of the underflow or overflow. The value in I is not the true exponent, but what was left in the exponent field of a floating point number after the interrupt.

[1]A program interrupt occurs asynchronously.

[2]The user exit routine may supply an alternate answer for the setting of the result register. This is accomplished by placing a value for D in the user-exit routine. Although the interrupt may be caused by a long or short floating-point operation, the user-exit routine need not be concerned with this. The user-exit routine should always set a REAL*8 variable and the FORTRAN library will load short or long depending upon the floating-point operation that caused the interrupt.

[3]For floating-point divide check, the contents of the result register is shown in the message.

[4]The user-exit routine does not have the ability to change result registers after a fixed-point divide check.

Figure 52. Corrective Action After Program Interrupt Occurrence

Option Table Considerations

Figures 26, 27, and 28 in the section "Extended Error Handling Facility" describe the fields of the option table and list the default values for the contents of these fields.

When a user-written exit subroutine is to be executed for a given error condition, the programmer must enter the address of the routine into the option table entry associated with that error condition.

Addresses for user-exit subroutines cannot be entered into the option table entries during program installation. An installation may, however, construct an option table containing user-exit addresses and place that option table into the FORTRAN library. (Each address must be specified as a v-type address constant.) Use of this procedure, though, results in the inclusion in the load module of all such user-exit subroutines.

If the user-exit address is not specified in advance through the use of v-type address constants the programmer must issue a CALL ERRSET statement at execution time to insert an address into the option table that was created during program installation.

The programmer should be warned that altering an option table entry to allow "unlimited" error occurrence (specifying a number greater than 255) may cause a program to loop indefinitely.

Considerations for the Library Without Extended Error Handling Facility

When the extended error handling facility is not chosen, execution terminates after the first occurrence of an error, unless it is one caused by divide check, exponent underflow, or exponent overflow. The messages for errors 215, 216, 218, 221-225, and 240-301 are the same as those with the extended error handling facility. The other error messages are of the form IHN*xxx* or IHO*xxx* with no text.

Without the facility, ERRMON becomes an entry point to the traceback routine. User programs that call the error monitor do not have to be altered. The error message will be printed with a traceback map and execution will terminate.

Note, too, that if the facility is not selected, the ERRTRA, ERRSET, ERRSAV, and ERRSTR subprograms are assumed to be user supplied if they are called in a FORTRAN program.

Appendix E: Defining Execution-Time Files for Compatibility with OS

Your FORTRAN programs, running under CMS, can create files that are acceptable to OS or use files that were originally created under OS. For sequential, unlabelled tape files, this transfer can be made directly without any intervening conversion of storage medium. For sequential and direct access disk files the files must first be placed onto tape and then placed on the disks of the system that is to receive them. To make use of such files, you must use the RECFM, LRECL, and BLKSIZE options of the FILEDEF command as they would be used in the DD statement under OS. The information presented below is designed as an aid in understanding OS data set formats and, so, ease the conversion process.

The type of device you are using for your file and the way it is organized limit your choice of record format and, as a result, the way in which you specify logical record lengths and block sizes.

The value of LRECL for all fixed-length (F and FB) and undefined (U) records on all devices except tape is in the following range:

$1 \leq \text{LRECL} \leq$ *device-capacity*

The value of LRECL for all variable-length (V, VB, VS, and VBS) records on all devices except tape is in the following range:

$5 \leq \text{LRECL} \leq$ *device-capacity*

The value of LRECL for all types of records on a tape device is in the following range:

$18 \leq \text{LRECL} \leq 32760$

The value of BLKSIZE for fixed-length, unblocked (F) and undefined (U) records is determined as follows:

BLKSIZE = LRECL

The value of BLKSIZE for fixed-length, blocked (FB) records is determined as follows:

LRECL • *number-of-records* = BLKSIZE $\leq$ *device-capacity*

The value of BLKSIZE for all variable-length (V, VB, VS, and VBS) records

(LRECL • *number-of-records)* + 4 = BLKSIZE $\leq$ *device-capacity*

The values for *device-capacity* are listed in Figure 53.

Device Type	Device Capacity (Maximum BLKSIZE)
Terminal[1]	133
Direct Access	
2301	20483
2302	4984
2303	4892
2305	
Mod I	14136
Mod II	14660
2311	3625
2314	7294
2319	7294
3330	13030
Card Reader	80
Card Punch[1]	81
Printer[1]	
120 chars.	121
132 chars.	133
144 chars.	145
150 chars.	151

[1]For variable-length records, add 8 to the values shown.

Figure 53. Maximum BLKSIZE by Device Types

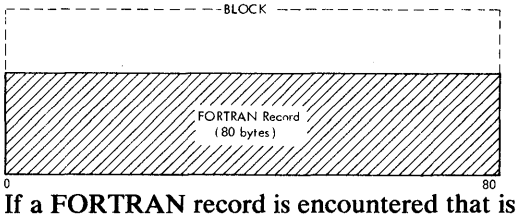
The description below outlines how these options are used in an OS environment and what effect they have on your files. Should you need more information, refer to the appropriate OS programmer's guide for your compiler. These books are listed in the preface of this book.

Formatted Records (Sequential and Direct Access)

- F (fixed-length, unblocked records)

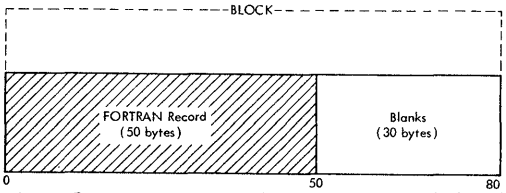
  All the records to be read or written are the same length, and a single READ or WRITE statement affects only one record at a time. The RECFM option specifies F and the BLOCK option specifies the length of the records. LRECL is not required.

  Example: Assume that all records are 80 bytes. RECFM is F and BLOCK is 80.



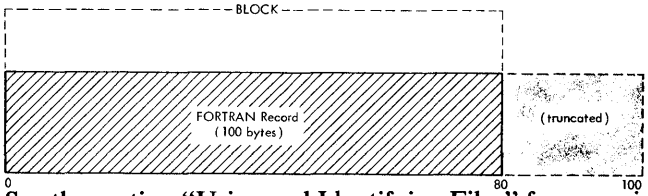
  If a FORTRAN record is encountered that is smaller than the block size, the unused portion of the block size is filled with blank characters.

  Example: Assume that a record is 50 bytes. RECFM is F and BLOCK is 80.



  If a FORTRAN record is encountered that is longer than the block size, the additional portion of the record is truncated and lost.

  Example: Assume that a record is 100 bytes. RECFM is F and BLOCK is 80.



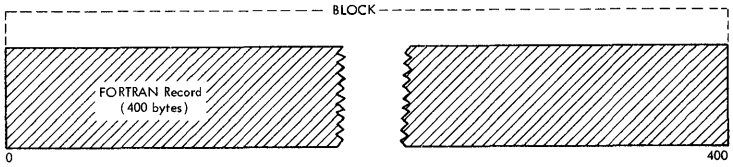
  See the section "Using and Identifying Files" for more information on using sequential and direct access files.

Formatted Records (sequential only)

- U (undefined-length records)

  The length of the records to be read or written is not defined; however, you must account for the longest record that may be encountered as specified by your FORMAT statement. A single READ or WRITE statement affects only one record at a time. The RECFM option specifies U and BLOCK specifies the length of the longest possible record. LRECL is not required. Any unused space is not read or written, and records that are too long are truncated.

Example: Assume that the longest record that can be encountered is 400 bytes. RECFM is U and BLOCK is 400.

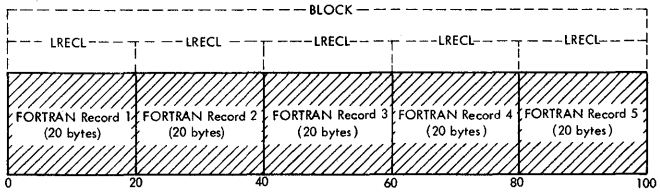


See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

- FB (fixed-length, blocked records)

  All the records to be read or written are the same length; however, a single READ or WRITE statement affects a group of records (that is, as many fixed-length records as will fit in the block size specified). The RECFM option specifies FB, LRECL specifies the size of the records, and BLOCK will determine the number of records in the group. It must be an exact multiple of LRECL.

  Example: Assume that all the records are 20 bytes and that 5 records are to be grouped together. RECFM is FB; LRECL is 20, and BLOCK is 100.



  As with fixed-length unblocked records, any FORTRAN records that are shorter or longer than LRECL or that do not meet the requirements of BLOCK are padded with blanks or truncated. See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

- FBS/FS (fixed-length, blocked and unblocked records with standard blocks)

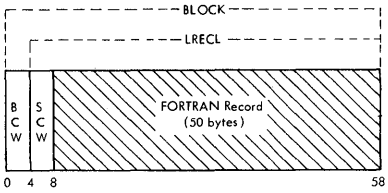
  All records are to be contained in standard blocks (that is, only the last record may be padded with blanks or truncated if it does not fit the block exactly). The FILEDEF options are specified in the same way as fixed-length, blocked and unblocked records.

- V (variable-length, unblocked records)

  All the records to be read or written are not the same length. A single READ or WRITE statement affects only one record at a time. The RECFM option specifies V, LRECL specifies the length of the longest record plus 4 bytes for a segment control word, and BLOCK specifies the value of LRECL plus an additional 4 bytes for a block control word. Since the records are varying lengths the segment and block control words inform the system of the length of the records involved. Each record has a segment control word and each block has a block

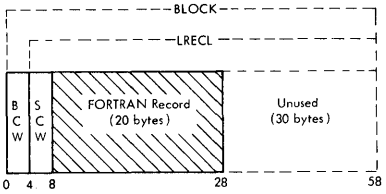
control word. For all file modes except 4 the block and segment control words are removed before the block is written.

Example: Assume that the longest record is 50 bytes. RECFM is V; LRECK is 54, and BLOCK is 58.



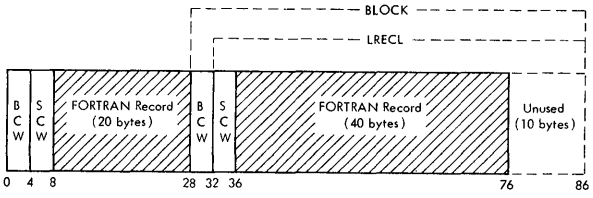
If a FORTRAN record is encountered that is shorter than the length specified in LRECL, the unused portion is ignored.

Example: Assume that a record of 20 bytes is encountered. RECFM is V; LRECL is 54, and BLOCK is 58.



Only the space required for the record is used since the block and segment control words contain information on the length and position of the record. The next record to be written will begin immediately after the record, not the block; the unused portion of the previous record is used for the following record.

Example: Assume the next record to be written is 40 bytes. RECFM is V; LRECL is 54, and BLOCK is 58.



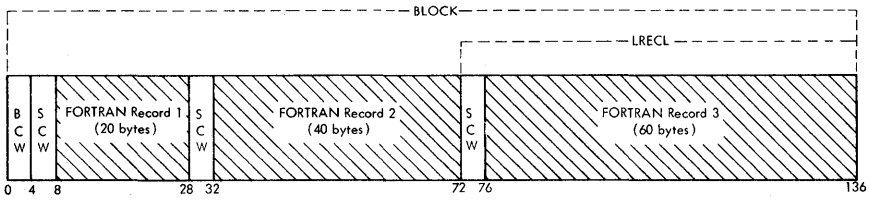
If a record exceeds the size of the block it is truncated. See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

- VB (variable-length, blocked records)

  All records to be read or written are not the same length. A single READ or WRITE statement affects a group of records (that is, as many variable-length records as will fit in the block size specified). The RECFM option specifies VB; LRECL specifies the length of the longest record plus 4 bytes for a segment control word, and BLOCK specifies a value longer than LRECL plus 4 additional bytes for a block control word. Remember, each record placed in the block requires 4 bytes for its segment control word. Since the block and segment control words contain information about the length and

position of the records each record occupies only the space it requires. Unused space at the end of the block is ignored. For all file modes, except 4, the block and segment control words are removed before the block is written.

Example: Assume that there are to be three records of 20, 40, and 60 bytes in a block. RECFM is VB; LRECL is 64, and BLOCK is 136 (the block size specifies the exact amount of space required to contain the records).



For efficient use of storage when you are using VB records, always do the following:

1. Specify a value for LRECL. If LRECL is omitted, only one record will be contained in each block.
2. Determine, in advance whenever possible, the exact size for the BLOCK option. Any unused space in the block is lost and any records that exceed the block size are truncated.

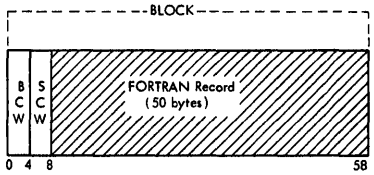
See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

Unformatted Records (sequential only)

- VS (variable-length, spanned records)

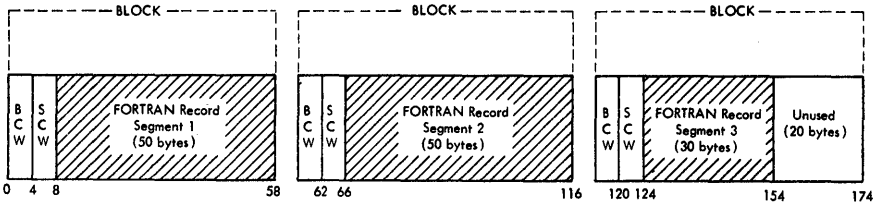
  All the records to be read or written are not the same length. A single READ or WRITE statement affects only one record at a time. The RECFM option specifies VS, and the BLOCK option specifies a pseudo-block size. LRECL is not required. The pseudo-block size may be smaller than, larger than, or equal to the length of any of the records used. Block and segment control words contain information on the length and position of a record or parts of a record in a block. For a record and its associated segment control word that is smaller than or equal to the block size minus its block control word, a single record occupies one block. Unused space is ignored. Spanned records can only be placed in a file with a file mode of 4.

  Example: Assume that a record of 50 bytes is equal to the pseudo-block size. RECFM is VS, and BLOCK is 58.



For a record that is larger than the block size minus it block control word, the record occupies (that is, spans) as many blocks as are necessary to contain it. Any unused space is ignored.

Example: Assume that a record of 130 bytes is larger than the pseudo-block size. RECFM is VS, and BLOCK is 58.

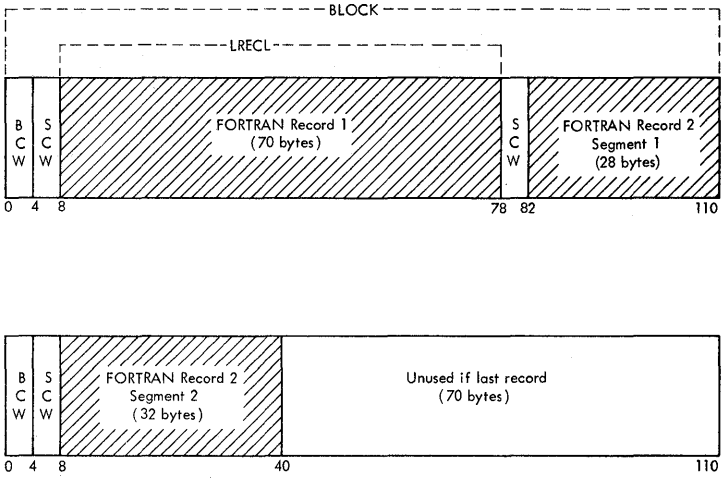


See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

- VBS (variable-length, blocked, spanned records)

  All records to be read or written are not the same length; however, a single READ or WRITE statement may affect more than one record at a time, depending upon the length of the records. The RECFM option specifies VBS; the LRECL option specifies the length of the longest record that may be encountered, and the BLOCK option specifies a value larger than the value of LRECL but not necessarily a multiple. The block size specified will contain as many records as will fit and may span the last record into the next block. Although block and segment control words are used, they need not be specifically accounted for in LRECL and BLOCK. Any unused space in a block containing the last record is ignored. Spanned records can only be placed in a file with a file mode of 4.

Example: Assume that there are two records of 70 and 60 bytes in a block. RECFM is VBS; LRECL is 70, and BLOCK is 110.



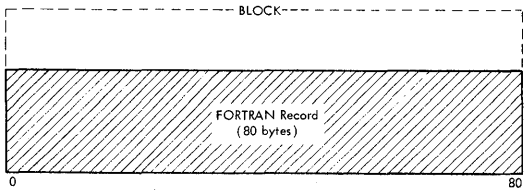
See the section "Identifying and Using Execution-time Files" for more information on using sequential files.

Unformatted Records (direct access only)

- F (fixed-length, unblocked records)

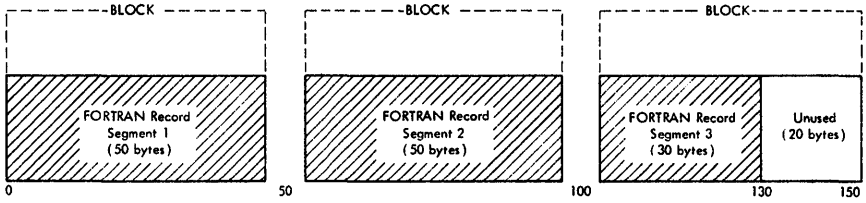
  The length of the records to be read or written is determined by the FORTRAN DEFINE FILE statement and cannot be changed with the LRECL option. A single READ or WRITE statement affects only one record at a time. The RECFM option specifies F, and BLOCK specifies a pseudo-block size, which may be smaller than, equal to, or larger than the length of the record specified in the DEFINE FILE statement. LRECL is not required. There are no block or segment records. Records that are smaller than or equal to the block size, occupy one block. Unused space is ignored.

  Example: Assume that a record of 80 bytes is equal to the block size. RECFM is F, and BLOCK is 80.



  If the record is larger than the block size, the record occupies as many blocks as are required to contain it.

Example: Assume that a record of 130 bytes is larger than the block size. RECFM is F, and BLOCK is 50.



See the section "Identifying and Using Execution-time Files" for more information on using direct access files.

Appendix F: Error Messages

The following messages are produced in response to entering an incorrect FORTRAN compiler command or when the command cannot be executed. The format of the messages is:

DMS *xxxnnn* E text of the message

where:

xxx - indicates the compiler in use

IGI - FORTRAN IV (G1) Compiler

IGK - Code and Go FORTRAN IV Compiler

IFE - FORTRAN IV (H Extended) Compiler

CON - SIFT Utility

nnn - is the message number

DMS *xxx* 001E NO FILENAME SPECIFIED

Explanation: You have not included a filename in the compiler command.

System Action: None

Programmer Response: Reissue the appropriate compiler command and specify a filename.

DMS *xxx* 002E FILE '*filename FORTRAN*' NOT FOUND

Explanation: The filename that you included in the compiler command does not correspond to the names of any of the files on your disks.

Supplemental Information: The variable *filename* in the text of the message indicates the name of the file that could not be found.

System Action: None

Programmer Response: Reissue the compiler command with an appropriate filename.

DMS *xxx* 003E INVALID OPTION '*option*'

Explanation: You have included an invalid option with your compiler command.

Supplemental Information: The variable *option* in the text of the message indicates the invalid option.

System Action: None

Programmer Response: Check the format of the appropriate compiler command and reissue the command with the correct option.

DMSIFE004W WARNING MESSAGES ISSUED

Explanation: The compiler has detected errors in your program and issued diagnostic messages whose highest severity level is 4.

System Action: None

Programmer Response: Check your terminal listing for any compiler diagnostic messages. Correct the errors in your program and recompile it.

DMS*xxx* 005E NO *option parameter* SPECIFIED

Explanation: You did not supply a required parameter for an option that was included with your compiler command.

Supplemental Information: The variable *option parameter* in the text of the message indicates the option that requires the parameter.

System Action: None

Programmer Response: Check the format of the appropriate compiler command and reissue the command with the correct parameter.

DMS*xxx* 006E NO READ/WRITE DISK ACCESSED

Explanation: Your virtual machine configuration does not include a read/write disk for this terminal session or you failed to specify a read/write disk in your ACCESS command following LOGIN.

System Action: None

Programmer Response: Issue an ACCESS command specifying a read/write disk.

DMS*xxx* 007E FILE '*filename* FORTRAN' IS NOT FIXED, 80 CHAR. RECORDS

Explanation: The FORTRAN source file that you specified in the compiler command does not contain fixed length records of 80 characters. The command cannot be executed.

Supplemental Information: The variable *filename* in the text of the message indicates the name of the FORTRAN file that is in error.

System Action: None

Programmer Response: You must reformat your file into the correct record length.

DMSIFE008W ERROR MESSAGES ISSUED

Explanation: The compiler has detected errors in your program and issued diagnostic messages whose highest severity level is 8.

System Action: None

Programmer Response: Check your terminal listing for any compiler diagnostic messages. Correct the errors in your program and recompile it.

DMSIFE012W SEVERE ERROR MESSAGES ISSUED

Explanation: The compiler has detected errors in your program and issued diagnostic messages whose highest severity level is 12.

System Action: None

Programmer Response: Check your terminal listing for any compiler diagnostic messages. Correct the errors in your program and recompile it.

DMSIFE016W UNRECOVERABLE ERROR MESSAGES ISSUED

Explanation: The compiler has detected errors in your program and issued diagnostic messages whose highest severity level is 16.

System Action: None

Programmer Response: Check your Terminal listing for any compiler diagnostic messages. Correct the errors in your program and recompile it.

DMS*xxx*034E FILE '*filename* FORTRAN' IS NOT FIXED LENGTH

Explanation: The file that you specified in the compiler command does not have fixed length records.

Supplemental Information: The variables *filename* and in the text of the message indicates the name of file in error.

System Action: None

Programmer Response: You must reformat your file into the correct record length.

DMS*xxx*038E FILE ID CONFLICT FOR DDNAME '*ddname*'

Explanation: You issued a FILEDEF command that conflicts with an existing FILEDEF for the ddname specified.

Supplemental Information: The variable *ddname* in the text of the message indicates the ddname in error.

System Action: None

Programmer Response: Reissue the FILEDEF command with an appropriate ddname.

DMSIGKE045E PROGRA NOT EXECUTED DUE TO SEVERITY CODE

Explanation: This message applies only to the Code and Go FORTRAN IV processor. Errors of sufficient severity to prevent compilation or a successful execution were present in your source program.

System Action: The compilation was terminated or the compilation was completed but execution was not begun.

Programmer Response: Correct the errors in your source program and recompile and execute it.

DMS*xxx*052E MORE THAN 100 CHARS, OPTIONS SPECIFIED

Explanation: The string of options that you specified with your compiler command exceeded 100 characters in length.

System Action: None

Programmer Response: Reissue your compiler command with fewer options specified.

DMS*xxx*070E INVALID PARAMETER *'parameter'*

Explanation: You specified an invalid parameter for an option in the compiler command.

Supplemental Information: The variable *parameter* in the text of the message indicates the invalid parameter.

System Action: None

Programmer Response: Check the format of the option with its appropriate parameters and reissue the command with the correct parameter.

DMS*xxx*075E DEVICE *device name* ILLEGAL FOR INPUT/OUTPUT

Explanation: The device specified in your FILEDEF command cannot be used for the input or output operation that is requested in your program. For example, you have tried to read data from the printer.

Supplemental Information: The variable *device name* in the text of the message indicates the incorrect device that was specified. In addition, the text will indicate whether an input or an output operation was requested.

System Action: None

Programmer Response: Reissue your FILEDEF command specifying an appropriate device for the desired input or output operation.

DMSCON300E GOFORT MISSING OR MISSPELLED

Explanation: You omitted or misspelled to GOFORT portion of the CONVERT command.

System Action: None

Programmer Response: Reissue the CONVERT command with GOFORT spelled correctly.

DMSCON301E FILENAME NOT FOUND

Explanation: The input filename that you included in the CONVERT command does not correspond to the names of any of the files on your disks.

System Action: None

Programmer Response: Reissue the CONVERT command with an appropriate filename.

DMSCON302E FILENAME '*filename*' WRONG RECORD LENGTH

Explanation: The output file that you identified in the CONVERT command does not have a RECFM of F or a BLOCK of 80.

Supplemental Information: The variable *filename* in the text of the message indicates the filename in error.

System Action: None

Programmer Response: Reissue the CONVERT command with a filename that has the appropriate characteristics or redefine the filename that you specified.

DMSCON303E FILENAME FOR SIFT OUTPUT OMITTED

Explanation: You did not specify an output filename in the CONVERT command.

System Action: None

Programmer Response: Reissue the CONVERT command specifying an output filename.

DMSCON304E OUT FILE CANNOT BE THE SAME AS IN

Explanation: You specified the same filename for the input and output files in the CONVERT command.

System Action: None

Programmer Response: Reissue the CONVERT command specifying appropriate filenames for your input and output files.

DMSCON305E NO READ/WRITE DISK ACCESSED

Explanation: Your virtual machine configuration does not include a read/write disk for this terminal session or you failed to specify a read/write disk in your ACCESS command following LOGIN.

System Action: None

Programmer Response: Issue an ACCESS command specifying a read/write disk.

DMSCON306E FILE CONFLICT FOR DDNAME 'ddname'

Explanation: You issued a FILEDEF command that conflicts with an existing FILEDEF for the ddname specified.

Supplemental Information: The variable *ddname* in the text of the message indicates the ddname in error.

System Action: None

Programmer Response: Reissue the FILEDEF command with an appropraite ddname.

DMSCON307E DEVICE *device name* ILLEGAL FOR INPUT

Explanation: The device specified in your FILEDEF command cannot be used for the input operation that is requested by the CONVERT command.

Supplemental Information: The variable *device name* in the text of the message indicates the incorrect device that was specified.

System Action: None

Programmer Response: Reissue your FILEDEF command specifying an appropriate input device for the CONVERT command.

Glossary

Batch processing. A method of using a computer system that enters sets of programs or data sequentially. One set is processed before the next is begun.

Dominance relationships. The logical relationships that exist in a program where parts of the program dominate other parts. That is, logic always flows to a part of a program through another part.

Pre-defined files. Files that are defined by the system for the user and are available any time the user wants them.

Problem solving programmer. A programmer who writes, debugs, and executes relatively short programs at a terminal.

Production Programmer. A programmer who debugs components of a large program on-line before running the program through a production-oriented processor.

Self-prompting. A method of coding programs whereby a user notifies himself at the terminal that the program is ready to accept input from the terminal.

System administrator. The person in the computing center who is responsible for the system or who assists terminal users in their use of the system.

Time-sharing. A method of using a computing system that allows a number of users to execute programs concurrently and to interact with the programs during their execution.

User-defined files. Files that are used or created by a user's program and which the user defines for himself.

Virtual machine. A simulated computer created by VM/370 that offers the user all the facilities of a real computer and that operates on a shared-time basis with the other users of the system.

IBM VM/370 (CMS) Terminal User's Guide
for FORTRAN IV Program Products
SC28-6891-1

Reader's Comment Form

Your comments about this publication will help us to improve it for you. Comment in the space below, giving specific page and paragraph references whenever possible. All comments become the property of IBM.

Please do not use this form to ask technical questions about IBM systems and programs or to request copies of publications. Rather, direct such questions or requests to your local IBM representative.

If you would like a reply, please provide your name, job title, and business address (including ZIP code).

Fold on two lines, staple, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Fold and Staple

First Class Permit
Number 439
Palo Alto, California

Business Reply Mail

No postage necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Corporation
System Development Division
LDF Publishing—Department J04
1501 California Avenue
Palo Alto, California 94304

Fold and Staple

CMS TUG for FORTRAN IV Program Products Printed in U.S.A. SC28-6891-1

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

SC28-6891-1

CMS TUG for FORTRAN IV Program Products Printed in U.S.A. SC28-6891-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)