MULTICS FORTRAN
USER'S GUIDE

SUBJECT

Information Concerning the Creation and Execution of FORTRAN Programs in a Multics System, with Suggestions for Efficient Coding, Use of the I/O System, and an Introduction to Basic Multics Concepts

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CC70-01

December 1979

Honeywell

PREFACE

The purpose of this manual is to supplement Multics FORTRAN, Order No. AT58.

Anyone faced with the prospect of learning to use an unfamiliar computer system is likely to experience some frustration in trying to get information out of the manuals that are supposed to explain it all. The inexperienced or occasional user is often at a loss for where to start understanding it all, especially since the manual explaining it all seems to assume everything. You want to know where there is a manual explaining how to use the manual that is supposed to explain it all.

The FORTRAN User's Guide is written in the hope that all of you who want to write FORTRAN programs on the Multics system can get answers to basic questions both about the system and about the the FORTRAN dialect embodied on it.

If you are new to the system, whatever your level of sophistication as a programmer, the first section, "Introduction to Multics," provides a general overview of the system from the standpoint of FORTRAN programming. You are strongly encouraged to read through this section carefully before reading any other part of the manual.

Sections are so designed as to make them independent of each other. Depending on what you want to know, you can read the rest of the manual in any order you choose.

The FORTRAN language on Multics is a superset of ANSI Standard FORTRAN, 1966. As such it contains features not defined by the standard of 1966, either in the form of extensions to the standard (especially in regard to Input/Output processing), or in the form of nonstandard features that are familiar to most users of other systems. In some cases, Multics FORTRAN extends the standard of 1966 to meet that of 1977.

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1983

File No.: 1L13 CC70-00

The FORTRAN language on Multics is a superset of ANSI Standard FORTRAN, 1966. As such, it contains features not defined by the standard of 1966, either in the form of extensions to the standard (especially in regard to Input/Output processing) or in the form of nonstandard features that are familiar to most users of other systems. In many cases, Multics FORTRAN now extends the standard of 1966 to meet that of 1977.

Some but not all of the features of FORTRAN 77 are available only if the program is compiled with the ansi77 option in effect. Only those features that are incompatible with the ansi66 interpretation are under control of the ansi77 option. Almost all of the FORTRAN 77 standard features have now been inplemented.

For a formal description of the FORTRAN language embodied on Multics, see the Multics FORTRAN manual, Order No. AT58. The user is assumed to have a working knowledge of FORTRAN. No attempt is made to provide instruction in the writing of FORTRAN programs, although some suggestions are offered about how to make them more efficient.

Throughout this manual, references are made to portions of the MPM. For convenience, these references are shortened as follows:

Document	Referred to in Text as
Reference Guide (Order No. AG91)	MPM Reference Guide
Introduction to Programming on Multics (Order No. AG90)	Introductory Users' Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Subroutine (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide
Communications Input/Output (Order No. CC92)	MPM Communications I/O

Significant Changes in CC70-01C

Implementation of Large Arrays and Very Large Arrays
   Addition of managed storage
Addition of -long_profile control argument
Several machine-dependent global optimizations now performed by compiler:
      Global Pointer Register Use
      Global Index Register
      Processor Instruction Fetch Padding

*

For purposes of clarity and ease of use, the MPM set has been reorganized. The six former MPM manuals, the Tools manual, and the RCP Users' Guide have been consolidated into a new set of three manuals.

Multics Programmer's Reference Manual (AG91)
   contains all the reference material from the former eight manuals.

Multics Commands and Active Functions (AG92)
   contains all the commands and active functions from the former eight manuals.

Multics Subroutines and Input/Output Modules (AG93)
   contains all the subroutines and I/O modules from the former eight manuals.

The following manuals are obsolete:

Name	Order No.
MPM Peripheral Input/Output	AX49
MPM Subsystem Writers' Guide	AK92
Programming Tools	AZ03
MPM Communications I/O	CC92
Resource Control Users'Guide	CT38

References to these manuals still exist on pages not published with this addendum. When this manual is revised, the references in the text to the old manuals will be changed to reflect the new organization.

CONTENTS

		Page
Section 1	Introduction to Multics	1-1
	Directory Hierarchy	1-1
	Segments	1-1
	Directories	1-1
	Entries and Entrynames	1-2
	Pathnames	1-2
	Example of a Pathname	1-3
	Working Directory	1-3
	Absolute Pathname	1-3
	Relative Pathname	1-4
	Home Directory	1-4
	Directory Hierarchy = File System	1-4
	Pathnames vs. Entrynames	1-5
	Search Rules	1-5
	Commands and Command Level	1-5
	The Multics Programming Environment	1-7
	The Basic Data Structures	1-7
	The Stack	1-7
	Free Storage Region	1-8
	Managed Storage	1-8
	Load Modules, Object Segments, and Linking	1-8
	Load Module	1-8.1
	Object Segment	1-8.2
	The Linkage Section	1-9
	Reference Names and Entrypoint Names	1-10
	The Reference Name Table	1-10
	Consequences of Dynamic Linking	1-11
	The Main Program Concept	1-12
	Common Block References	1-12
	Permanent Common Blocks	1-14
	Common Statement	1-15
	Storage Classes	1-15
	Automatic Variables	1-17
	Program Units Compiled Together	1-17
	Quick Calls	1-18
	Separately Compiled Program Units	1-19.1

CONTENTS (cont)

		Page
	Failure to Initialize Automatic Variables	1-19.2
	Initialization	1-20
	Undefined Variables	1-20
	Save Statement	1-20
	Automatic Statement	1-21
	Data Statement	1-21
	Run Units	1-22
	Use of the Run Command	1-23
	Quit, Start, Release	1-23
	Pause, Start, Release, Stop	1-23.1
	Automatic Storage in Stack Frames	1-26
	Binding FORTRAN Programs	1-31
Section 2	Entering Your FORTRAN Source Program	2-1
	Login	2-1
	Creating a Source Segment	2-1
	Input Format	2-2
	Uppercase and Lowercase Letters	2-3
	Names in the FORTRAN Program	2-3
	Free-Form Format	2-3
	Comments	2-3
	Continuation Lines	2-5
	Semicolon	2-5
	Line Numbers	2-7
	Card-Image Format	2-7
	Comments	2-8
	Continuation Lines	2-8
	Line Numbers	2-8
Section 3	Compiling and Executing the FORTRAN Program	3-1
	Invoking the Compiler	3-1
	Error Diagnostics	3-2
	Control of Error Messages	3-4.1
	Language Options	3-5
	Subscript Checking	3-6
	Relocation	3-7
	Listing Segment	3-7
	Format of listing Segment	3-7
	Optimization	3-9
	Improving Program Speed	3-10
	Card-Image and Free-Form Source Programs	3-10.1
	Debugging	3-11
	Executing a FORTRAN Program	3-12
Section 4	Constraints	4-1

CONTENTS (cont)

		Page
	Length and Form of Records	4-1
	Files	4-2
	IO Transfer Limits	4-2
	Programs	4-2
	Statements and Line Numbers	4-2
	Arrays and Common Blocks	4-3
	Binder	4-3
	Stack Segment	4-3
	Normal Storage vs. Large Arrays and Very Large Arrays	4-4
	Large Arrays and Very Large Arrays	4-5
	Accuracy of Real Numbers	4-6
	Overflows in Integer Multiplications .	4-6
Section 5	Input/Output in Multics FORTRAN	5-1
	Introductory Comments	5-1
	Fundamentals of Input/Output . . .	5-1
	Implicit Connection	5-2
	The Use of Implicit Connection . .	5-4
	Input Data Transfers	5-4
	Output Data Transfers	5-4
	Explicit Connection	5-5
	Using the Open Statement	5-5
	What Is in This Subsection . . .	5-6
	Terminal Read/Write (Unit 0) . .	5-8.1
	Terminal Read (Units 5 and 41) .	5-9
	Terminal Write (Units 6 and 42)	5-9
	Formatted Sequential I/O to Storage System Files	5-10
	Unformatted Sequential I/O to Storage System Files	5-11
	Direct Access Formatted I/O to Storage System Files	5-12
	Direct Access Unformatted I/O to Storage System Files	5-13
	Binary Stream Files	5-15
	Connecting Nonstandard Units to the Terminal	5-16
	Connecting a Default Terminal Unit to a File	5-16
	Connecting 6 or 42 for Terminal Input, 5 or 41 for Terminal Output	5-17
	Connections to Tape Files . . .	5-17
	Using the Inquire Statement	5-18
	io_call attach for Device Independence	5-18.1

CONTENTS (cont)

		Page
	io_call open for Complete External Connection	5-20
	What's a I/O Switch?	5-22
Section 6	Conversion to Fortran 77	6-1
	FORTRAN 77 on Multics	6-1
	Conversions	6-2
	Character-Mode Variables in Common Blocks	6-2
	Equivalencing Character-Mode Data .	6-2
	Default Character-String Length . .	6-2
	Packed Character-String Layout . .	6-3
	Zero-Trip Do Loops	6-3
	Blank Lines	6-4
Appendix A	Debugging	A-1
	Specifications for Tic-Tac-Toe Program	A-1
	How the Program Works	A-1
	A Program to Play Tic-Tac-Toe . . .	A-3
	Script of Debugging Session	A-8
	Corrected Program to Play Tic-Tac-Toe	A-19
Appendix B	Optimization	B-1
	Local Optimizations	B-1
	Machine-Independent Local Optimizations	B-1
	Machine-Dependent Local Optimizations	B-2
	Quick Subprogram Call	B-2
	Implied Do-Loops	B-3
	Global Optimizations	B-4
	Machine-Independent Global Optimizations	B-4
	Non-Loop-Oriented Optimization . .	B-4
	Removal of Common Subexpressions	B-4
	Constant Propagation	B-6
	Loop-oriented Optimizations . .	B-7
	Removal of invariant expressions from loops	B-7
	Strength Reduction	B-8
	Test Replacement	B-9
	Removal of Dead Assignments . .	B-10
	Machine-Dependent Global Optimization	B-10
	Global Pointer Register Use . .	B-10
	Global Index Register	B-10.1

CONTENTS (cont)

		Page
	Processor Instruction Fetch Padding	B-10.1
	Pointers for Efficient Coding Style using the FORTRAN Optimizer	B-11
Appendix C	Compatibility with Non-FORTRAN Programs .	C-1
Appendix D	Error Messages	D-1
	Compile-Time Error Messages	D-1
	Runtime I/O Error Messages	D-20.1
Index	. . . .	i-1

ILLUSTRATIONS

Figure 1-1.	Directory Entries	1-2
Figure 1-2.	Dynamic Linking	1-9
Figure 1-3.	State of Stack	1-24
Figure 1-4.	Allocation of Stack Frames	1-27
Figure 1-5.	State of Stack in Program Compiling . . .	1-30
Figure 3-1.	Control Arguments	3-4.1

SECTION 1

INTRODUCTION TO MULTICS

This section provides basic information about the Multics environment. This information is background to an understanding of the Multics system, and is provided as a bare introduction to the new user. More introductory information is in the New Users' Introduction to Multics-Part I. If you are an experienced Multics user, you can skip this section and go directly to Section 2 or Section 3.

DIRECTORY HIERARCHY

Segments

The basic unit of storage in Multics is the segment. (A FORTRAN source program, the compiled object code, and the data it uses are all stored in segments.) The attributes of a segment--such as name and size--are recorded in the "catalogues" of the directory hierarchy.

Directories

The directory hierarchy is organized into a tree-structure of catalogues, or as they are called in Multics, directories. Directories contain descriptions of segments, of other directories, and of cross-references called links.

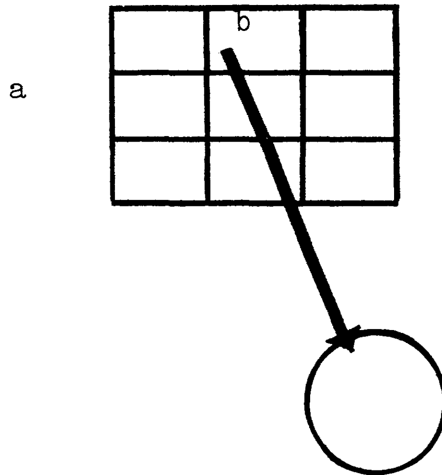


Figure 1-1. Directory Entries

In the figure above, "a" is a directory. The entryname of the segment, "b," appears as part of the directory entry, b, in directory a. The entryname is also associated with the contents of the segment named b - that is (in FORTRAN terms), the file stored in that segment.

Entries and Entrynames

The descriptions of segments and directories are called entries. An entry in the directory hierarchy is just a set of attributes, including a list of names called entrynames. Each entryname is unique in any one directory; no two entries in the same directory can have the same name. However, any entry can have several entrynames.

Examples of Entrynames:

```
test.list
test.fortran
file06
test
extremely_long_entrynames_occur
data.input.08/15/78-1430
```

Pathnames

To understand the form and function of pathnames it is useful to think of them first as the names of segments. Each segment is uniquely identified in the directory hierarchy by its pathname. A pathname is a series of entrynames separated by the

">" character. The sequence of these entrynames reflects the position in the hierarchy of the directory in which the segment is catalogued. An entryname is always relative to some directory in the hierarchy -- that is, to the directory containing the entry (sometimes called the "parent" directory). The final entryname in a pathname may designate either a directory or a segment. If the final entryname refers to a directory, the pathname is the name of a directory; otherwise it is the name of a segment.

Example of a Pathname

The pathname for a segment named x in Tom Smith's directory might be:

```
>udd>ProjA>TSmith>x
```

In the example above, the user has a directory that is given his name, TSmith; the entryname TSmith is in turn catalogued in the directory of the project Smith is registered under (ProjA), and the entryname ProjA is catalogued in the system directory udd, which contains an entry for each project. (The directory in which udd is contained is called the root; by convention this level of the hierarchy is omitted because it would appear at the beginning of every pathname.) The specified segment, x, is catalogued in the directory TSmith. Notice that the pathname represents a series of entries for directories, each within a parent directory down to the entry for the segment x. The entire pathname shows the "path" from the root to that segment.

Working Directory

The directory hierarchy, as so far described, may seem to be no more than a collection of different kinds of names. In fact, this is one important way to understand the hierarchy, because it affords both you and the system a way to locate data using names that are free of ambiguity.

ABSOLUTE PATHNAME

A pathname that begins with the ">" character is an absolute pathname, which means that the segment or directory it identifies is designated completely and unambiguously -- the name is all the information the system needs to locate the designated segment or directory.

RELATIVE PATHNAME

By convention, a pathname that does not begin with the ">" character is a _relative pathname_. It completely specifies the position of a segment relative to your _working directory_.

HOME DIRECTORY

At any time, your environment has one working directory. It is the directory whose absolute pathname the system prepends to any relative pathname you use. Your initial working directory is your user directory--usually the one with your name--and it is known as your _home directory_.

DIRECTORY HIERARCHY = FILE SYSTEM

Not only is the directory hierarchy a collection of names; it is also a filing system, and like any filing system it is an orderly and convenient way of arranging pieces of information. You can extend the hierarchy under your own directory, creating a hierarchy of your own within the larger one. This ability gives you a way to organize your work, to isolate part of it from the rest of the system, and ultimately to tailor for yourself the shape of your working environment. Finally, the hierarchy schema gives you a conceptual "space" to work in--for many programming tasks, it frees you from the necessity of thinking in terms of actual physical storage location. Usually, we say that we are "in" the working directory, and that some segment is in the such-and-such directory--you know it is actually in some physical location but where does not matter because the pathname fully defines the location for most purposes.

Pathnames vs. Entrynames

The name of a segment (segname) is a pathname or an entryname. An absolute pathname is all the information the system needs to locate the designated segment. The system interprets the relative pathname as an absolute pathname that includes your working directory; so as long as you know what working directory you are in, a relative pathname is unambiguous. The situation changes somewhat when the name of a segment is an entryname. An entryname alone is not enough to designate a particular segment or its position in the hierarchy, since there might be several segments with the same name but in different positions in the hierarchy. Given only an entryname, the system has to find out what segment you have in mind.

Search Rules

To find the segment, the system follows certain conventions known as the search rules. The system looks first in the list of segments you have used since logging in, and if the entryname appears there the search ends. Otherwise it looks in your working directory. If the entryname is not in your working directory, it looks in the system library where Multics commands are stored. Multics commands invoked by entrynames are found by the dynamic linker--for details on the linker, see "Reference Names and Entry point Names" later in this section.

The search rules permit you to invoke standard Multics commands by their entrynames. But if you have a program catalogued in your working directory under the same name as the entryname of a Multics system command, typing the entryname invokes the program in your working directory instead of the Multics command. It is possible to modify the search rules if you need to do so (see the set_search_rules and the add_search_paths commands in the MPM Commands).

Commands and Command Level

When a ready message is printed on your terminal, the system is ready to execute programs. The standard ready message consists of the letter "r" followed by the time of day and two numbers that reflect system resource usage, namely, central processor time and pages used.

```
r 19:44 4.375 3845
```

After the ready message appears on your terminal, the next line you type must begin with the name of a segment containing an executable procedure. In many cases this line includes a number of arguments, as:

segname arg1 arg2 ... arg*n*

The name of the segment, and the associated arguments (if any), order the system to execute the procedure designated by the name. An executable procedure is in a segment stored somewhere in the directory hierarchy. An example of such a procedure is a Multics system command, in which case the name of the segment and its associated arguments comprise a command line.

Since commands are invoked after the ready message is printed, the system at ready is said to be at command level. From command level you invoke a Multics system command by typing a command line on the terminal. The basic form of a command line is:

command_name <arguments>

The command name is the entryname of a segment that contains the object code of a procedure, and the arguments consist of one or more character strings. These arguments may be of two types: required arguments (simply called arguments) and optional arguments (called control arguments). A control argument is an optional parameter that may specify a variation on the basic action performed by a command. Arguments of both types are separated from the command name and from each other by blanks or tabs. (For information on the argument of the fortran command, see "Invoking the Compiler," in Section 3, and for a list of control arguments that may be supplied with the fortran command, see "Control Arguments," in Section 3.)

THE MULTICS PROGRAMMING ENVIRONMENT

The remainder of this section describes the fundamental data structures and the system conventions for dealing with these data structures that together go to make up the Multics programming environment. Because the Multics environment is unique among computer systems, you may find it confusing at the outset, especially because of two important features: dynamic linking and the use of a stack segment as storage for program variables. Both these topics are discussed below in such detail as is needed to make the material clear from the perspective of the FORTRAN programmer; if you are interested in learning more about Multics, refer to the Programmer's Reference Manual (AG91).

The Basic Data Structures

THE STACK

Three large storage regions underlie the Multics programming environment--the stack segment, the free storage area, and managed storage. The stack segment consists of a header (the details are in "Object Segment Format," in Section 4 of the Programmer's Reference Manual) followed by a series of stack frames. A new stack frame is added to the stack whenever a compilation unit--that is, a number of program units compiled into one object segment--is called, and removed from the stack (that is, completely discarded) when the compilation unit returns to its caller. (The stack frames are entirely managed by the system, and there is no need for you to take any action with regard to them.)

Each stack frame contains two kinds of information: control information and program-specific information. The control information, which consists of such items as stack threads and frame size, is not needed for programming. The program-specific information, which includes the name of the associated program and storage for certain classes of variables, may be of interest to you either for debugging or as an aid in determining how to assign variables to the different FORTRAN storage classes (see "Storage Classes," below).

FREE STORAGE REGION

The free storage region is a large storage area used both by the system and by the programmer. It contains linkage sections, the reference name table (RNT), and other system data (see "Linkage Section" and "Reference Name Table" below). The user data in the free storage region consists of common blocks and other FORTRAN variables that are allocated static storage. The methods of forcing variables into the various storage regions and the reasons for choosing one or another region are described below under "Storage Classes."

MANAGED STORAGE

Managed storage is used to hold Large Arrays and Very Large Arrays in automatic, static, and common. This storage is allocated on demand and freed on request. Managed storage used to hold automatic storage is freed when the corresponding stack frame is freed, managed storage used to hold static storage is filed when the program is terminated or deleted, and managed storage used to hold common is freed when a dev command for that common block is executed.

The process directory is normally used to contain managed storage, but the directory that will hold managed storage can be specified by setting the variable "fsm_dir_dir_path" in either the your per-process value segment or your permanent value segment to the pathname of the directory that will hold managed storage. This enables you to use some other directory to get managed storage quota when you do not have a particularly high process directory quota.

Load Modules, Object Segments, and Linking

A standard Multics object segment is the segment created by the compiler. It contains the compiled program, varying amounts of symbolic information, the initial values of variables, and the information necessary to enable the system to resolve at run time external addresses that the compiler itself cannot resolve. An external address, in this context, is the address of an object outside the segment in which a call originates. The resolution of external addresses at run time, whether the reference is to a common block or to another program, is what is known as dynamic linking. What makes it dynamic is that the system makes no attempt to learn the address until the external object is actually referenced during the execution of the program. Roughly speaking, an external reference is a call from one segment--however many program units it contains--to another segment. Calls between program units in a single segment are, in contrast, "internal."

LOAD MODULE

Most other computer systems employ a utility program called a linkage editor (or linking loader) to resolve all addresses before execution begins, creating what is generally called a load module. A typical load module contains executable code, local variables, common blocks, and varying amounts of symbolic debugging information. When you encounter Multics for the first time, you may wonder: "How do I load my program?" and "How do I figure out all the addresses in core?" The structure of Multics makes these questions misleading, as the rest of this section will make clear. For now, it is enough to say that in Multics, any object segment is "loaded" as soon as it exists, and that there is no core.

The main difference between the execution of a FORTRAN program embodied in a load module and a FORTRAN program embodied in Multics is the allocation (and as a result, initialization) of storage. The dynamic resolution of external addresses as such is probably not so important to FORTRAN users, in that the effect is nearly always the same whether resolution occurs at link-edit time or dynamically as in Multics. In what follows below, linking is discussed in somewhat fuller detail. If you are satisfied by the explanation of dynamic linking already given, you can skip ahead to "Consequences of Dynamic Linking," below, where essential information about linking appears.

OBJECT SEGMENT

The FORTRAN compiler creates an object segment whose contents the hardware uses directly. It is not necessary for you to make a copy that is loaded into main memory before execution (a "core image"). No code relocation is required, and you do not patch the code with the addresses of external objects (such as subprograms) unknown at compile time. In particular, in contrast to some other systems, there is no need to find space in main memory and put your program (i.e., load module) into it. The Multics virtual memory means that an object segment (if it exists) can be considered to be in main memory all the time.

The executable code in the Multics object segment can be shared in real time, which permits many users to execute the same program at the same time. What makes this sharing of executable code possible is that the code is isolated from data that cannot readily be shared, such as the values of variables and the addresses of external objects. The addresses of external objects cannot be shared because in Multics these addresses vary from user to user. All such information is stored separately for each user and remains stable for the life of the run, and even beyond. (See "Consequences of Dynamic Linking" below.") Since the actual addresses are unknown at compile time, the compiler generates a list of place-holders (known as the linkage section of the object segment, described further below). Each place-holder contains two items for each address, corresponding to the two parts of the actual address. The first item is a _fault tag_ that causes the program to stop running and sends a call to the operating system for action (a fault) when the address is referenced during the execution of the program. The second is a pointer to information elsewhere in the object segment itself, describing in symbolic form what the program references. When the central processor reaches an unresolved address during the execution of a program, execution halts temporarily and a call goes to a system module known as the _linker_. The linker transforms the symbolic address referenced in the program into a real address, and execution of the program resumes just as if the address had always been in the place held for it. (See Figure 1-2 for a graphic representation.) All subsequent references to that address require no further intervention from the system because the address has been written into the place reserved for it.

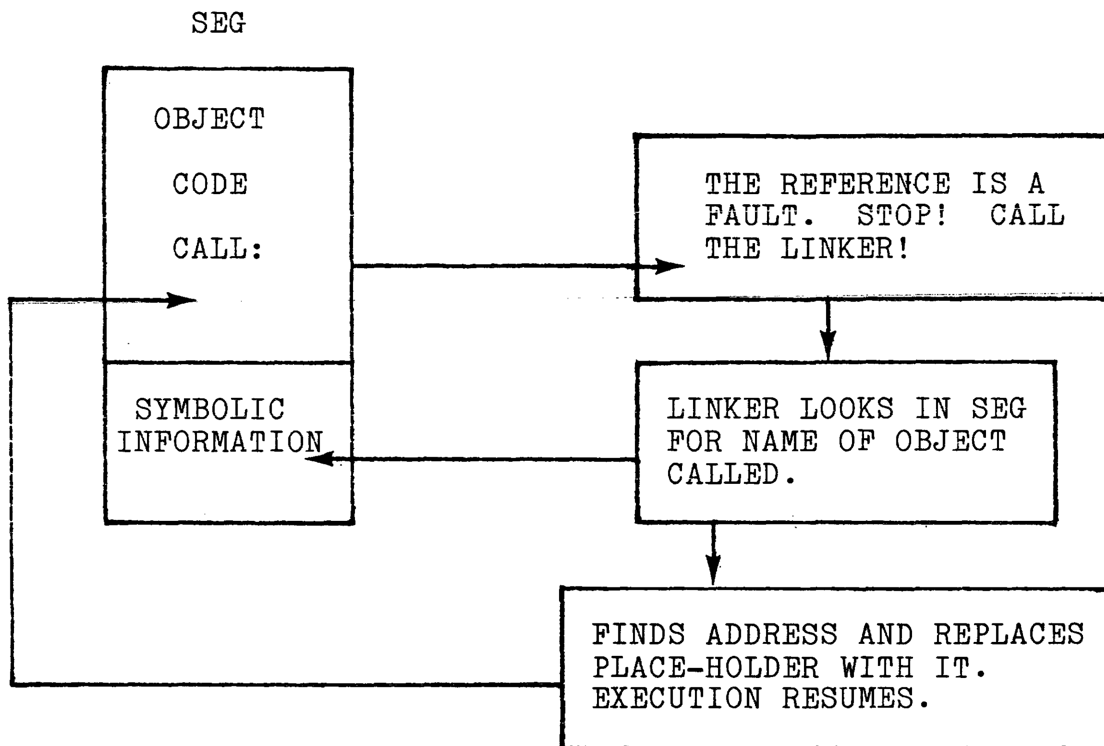


Figure 1-2. Dynamic Linking

The fault just described is called a linkage fault, and the process of resolving addresses through such faults is known as dynamic linking. It is dynamic because the resolution of external addresses is left until the last possible moment; that is, until execution actually reaches the external reference in the program.

THE LINKAGE SECTION

The list of place-holders for external addresses is the linkage section of the object segment. Since in the course of execution the linker must fill in place-holders with real addresses, which vary user by user, the linkage section is copied into the free storage region before execution of the program. When the external address is actually resolved, the linker writes the address into the appropriate place-holder in your copy of the linkage section. Each of these place-holders is known as a link. (The unfortunate use of the word link in so many different contexts in Multics is a regrettable accident of history; however, it is usually possible to determine unambiguously from context what kind of link is implied.) Whenever one program is called in your environment for the first time, its linkage

section is copied into free storage. This goes for the first time one program references another program as well. The linkage section of the second program is copied into free storage as a result of the linkage fault. (For more detailed information on the object segment and on linking, see the MPM Reference Guide.)

Reference Names and Entrypoint Names

The linker must be able to change a symbolic address into a real address of the form required by the hardware. Since the hardware uses a two-dimensional address consisting of segment number and offset within the segment (word number), the linker has to write into the place-holder an address in that form. Each link in the linkage section of the object segment contains a pointer to a symbolic name recorded elsewhere in the object segment (in the symbol section). This symbolic name is what the linker uses in finding the real address that it ultimately writes into the place-holder.

This symbolic name is in two parts: a reference name (which identifies the segment) and an entrypoint name (which tells where, in terms of offset location, the segment is entered). The linker takes the reference name and searches a set of directories for an entry with a matching name. If it finds the reference name, the linker then searches the corresponding segment for the entrypoint with the specified entrypoint name. If both searches are successful, the appropriate address is written into the link in the copy of the linkage section. (The link is said to be "snapped.") If either search fails, the linkage fault also fails.

In Multics the reference name and entrypoint name are combined, with a dollar sign character ($) between them as in hcs_$make_seg. In this case, hcs_ is the reference name and make_seg is the entrypoint name. If an external reference contains no dollar sign, as in ioa_, both reference name and entrypoint name are understood to be the same. As far as the linker is concerned, ioa_ is the same thing as ioa_$ioa_.

The Reference Name Table

Since searching directories is a fairly expensive task, the Multics system keeps track of all the reference names that the linker has successfully matched for you. These names and the necessary identification to locate the associated segment are held in a special table (allocated in your free storage region) called the reference name table (RNT). By first searching the RNT, the linker significantly reduces the search time needed to locate a segment referenced in a program, because if you have already referenced the segment elsewhere it will appear in the RNT. This preliminary search of the RNT is the default and standard action in Multics.

The search rules specify a list of directories to be searched by the linker and the order in which they are to be searched. In fact, the search rules may contain one thing that is not a directory--the RNT. By specifying the search rules appropriately, you can control the order in which directories are searched, as well as what directories are searched.

When a reference name is placed in the RNT and associated with a segment, we say that the name is initiated for that segment. The initiate command (see the MPM Commands manual) does just that--it explicitly associates a reference name (or names) with a specified segment. The set of names in the RNT refer to the initiated segments of your environment. (It is also known as the table of initiated segments.)

Names can be removed from the RNT by use of the terminate_refname command. Terminating is just the inverse of initiating.

The search rules in force can be manipulated by the three search rule commands: set_search_rules, add_search_rules, and print_search_rules. For more information on these commands, see the MPM Commands manual.

Consequences of Dynamic Linking

One interesting consequence of the reference name table strategy is that after a reference name is associated with a particular segment, all subsequent references to that name are resolved to that segment. The result of an external reference may be different from what you expect if one program calls another program by a given name--say, rollow--and there is already another program associated with the name rollow. There are several ways in which such an ambiguous reference might occur. The system, which uses these same mechanisms and shares the RNT with you, may have used the name already, or the association may have been made explicitly outside the calling program (see the initiate command in the MPM Commands), or the association may have been made implicitly by another program executed earlier and unknown to you, or which you have forgotten. This unexpected and usually unwanted situation can be avoided by the manipulation of search rules or by the use of the run command or the where command.

Usually load modules do not allow such ambiguous references, so if you are new to Multics you should take precautions to account for, and remove, potentially misleading references in advance of the program run.

The Main Program Concept

The Multics command processor (the system module that deals with commands) uses the linker to find the address of the entrypoint in the program the command calls. The command processor is subject to the same conventions as apply when one of your programs references any other program, i.e., the dollar sign convention, search of the RNT (if the search rules so specify), and initiation of the associated reference name at command level. In addition, a fourth convention applies at command level: if there is no dollar sign character in the command name (the name of your FORTRAN program, for example, rollow), and if there is no entrypoint a in segment a (that is, the rollow$rollow convention is not satisfied), then the segment is searched for an entrypoint with the name main_ (in this case rollow$main_). Multics system commands are understood to be of the form rollow$rollow. User programs are invoked as commands, but the naming is as follows: The FORTRAN compiler sets up entrypoint names in the object segment in such a way that the main unit in the program has the entrypoint name main_. Since a main program must be the first unit in a FORTRAN source program, there can be at most one such in any compilation. (If the first program is a subroutine or block data program, no main_ entrypoint name is created.) Note that two main programs originally compiled apart cannot be recompiled together since an object segment may contain only one main program. You could compile a main program and some or all of its accompanying subroutines independently, deciding later to combine them in one object segment (either by copying the source text into one segment, or by binding the compiled object code, for which see "Binding FORTRAN Programs," below). There would be no reason to compile or bind different main programs together, however.

> NOTE: the name main_ is reserved for system use, and you cannot use it as your own name for a program or as any other kind of symbolic name.

This main_ mechanism means that you can give the segment containing a FORTRAN program any other name you wish, or rename such a segment at your discretion, and the linker will still find the main entrypoint without the need of a dollar sign entrypoint name.

Common Block References

When a FORTRAN program references a common block, the standard linkage mechanism resolves the reference, but a special link is placed in the linkage section and the search rules are bypassed unless there is a "$" in the name. The linker searches the list of external variables currently defined in your environment.

Common blocks, as they are called in FORTRAN, in Multics terms are external variables. External variables are a special class of variable-allocated storage in the free storage region. The name of the external variable corresponding to a common block is identical to the name of the common block declared in a FORTRAN program. The amount of storage allocated for normal common blocks ~~can be up to 16 MW if the Very Large Array compilation feature is~~ selected; otherwise, common block length is limited to nearly the size of a segment (255K). In any case, the size of a permanent common block is limited to the size of a segment (255K). Common blocks in Multics are either permanent-named common blocks with a name that ends in the "$" character (a dollar sign name)--such a common block must exist before a program references it--or common blocks that are allocated and initialized at run time as specified in the program.

In many FORTRAN implementations on other systems, programs get one core load per run, with the result that both local and common storage are released at the end of the run. In the Multics system, normal (not permanent) common storage remains allocated until it is explicitly released. You might expect new common storage to be allocated at every run. For a discussion of the problems that arise and how to control them, see "Run Units," below.

The first program unit that references a common block should be compiled or bound with the block data subprogram that initializes the common block; otherwise there is no guarantee that the block is initialized correctly. Only a block data subprogram may contain data statements to initialize the variables in a common block. Data statements for common variables in other subprograms are in error.

The initial data placed in a common block when it is allocated depends on the declaration of the common block in the block data subprogram, assuming there is one. If there is no block data subprogram associated with the link that triggers allocation of a common block, the entire block of storage is initialized to zeroes. In order to associate a block data subprogram with a program in which common variables are referenced, you have a choice of three options. First, you can compile the program with the block data subprogram, that is, from the same source segment. Second, you can bind the program with the block data subprogram (see "Binding FORTRAN Programs," below). Third, you can use the set_fortran_common command to explicitly initialize (and, if necessary, allocate) the common block.

The set_fortran_common command is given the names of objects segments as arguments. Every object segment that is part of the FORTRAN run should be supplied on the command line to ensure that the common blocks are properly initialized. The command searches the segment for any common block definitions. For all such that it finds, it allocates and initializes the common blocks as appropriate.

If a common block is created as a result of action by the dynamic linker (that is, if it is not a preexisting named command block), all further references to the common block must be consistent with the first reference. For instance, if the first reference is from a program that declares only some of the variables in the common block, storage will be allocated only for these variables. Subsequently executed programs that reference the same common block cannot declare more variables than declared in the first program. If a second program is run in which other variables in the common block are declared, an error occurs because the program may attempt to reference beyond the storage allocated for the common block. The linker refuses to resolve the subsequent linkage fault.

To get around this problem, the simplest thing to do is to use identical declarations for one common block in all the program units that reference it. Another solution is to use the set_fortran_common command to specify the program for which the largest common block is declared. The set_fortran_common command should then be issued before any such FORTRAN programs are run.

PERMANENT COMMON BLOCKS

Permanent segments used as common blocks are identified in programs by a dollar-sign character ($) in the names of the common blocks and are located by the Multics linkage mechanism. Thus a segment called "name" in the storage system would be referred to as "name$" by a FORTRAN program.

Multics FORTRAN allows programs to reference permanent segments in the storage system directly rather than as files accessed by input/output statements. In many applications, this feature means a significant gain in performance. To make a direct reference to a permanent common block, the dollar-sign character ($) is used in the name of the common block, as described above. When the linker is asked to resolve an external reference to such a common block, the block is located through the search rules. If the name consists of a single name followed by a dollar sign (for example, name$), the entire segment associated with the reference name is treated as the common block. That is, the common block starts at offset zero in the segment. When the common block name has two components (for example, name$first), the segment must have an already defined entrypoint, but the standard linker conventions generate the real value for the offset defined by first.

Data in permanent segments must be generated by a FORTRAN program in order to comply with the internal format requirements for FORTRAN data.

Note that when information is stored in a permanent common block, the segment will continue to show a bit count of zero until the bit count is updated. To find out how much storage a segment is actually using, check the record count by using either the -records_used control argument with the status command or the -record control argument with the list command.

This page intentionally left blank.

COMMON STATEMENT

Use of the common statement causes variables to reside in common storage. Variables in each common block are ordered within the block in the order in which they are listed in the common statements of the subprogram. Double-precision and complex variables are always allocated on an even word boundary. As a result, a word of fill may precede a variable in the common block. The declarative statements used to define the common block need not be the same, but they must specify the same total amount of storage, including any fill added by the compiler. If the modes of the variables do not match, you must be aware of the mapping you obtain and the possible bad effects on the correctness of your program should you err, since the FORTRAN language generally requires that the modes match.

Initialization of unlabeled common is not permitted. Different subprograms may define different lengths for unlabeled common. Different subprograms must define identical lengths for the same named common block.

Storage Classes

There are basically four kinds of storage available to FORTRAN programmers (other than files used through input/output statements). These are:

automatic storage
static storage
normal common storage
permanent common storage

Automatic storage is normally allocated in the stack frame associated with a program. It is initialized when the stack frame itself is allocated, and again each time new storage for those variables is required; that is, for every new invocation of the associated program unit or units. It is the default storage class in Multics FORTRAN for local variables. A variable not declared to be in common storage is in local storage by default, and local storage is automatic by default. Local variables can be explicitly declared automatic with the automatic statement. Automatic variables are undefined when the corresponding program unit completes execution.

When Large Array or Very Large Array automatic storage is used, storage for arrays is allocated in managed storage, with pointers to this storage left in the stack frame. In such a case, that stack frame level (as known by the stack pointer) "owns" that managed storage. When the stack frame is released, the managed storage is also released.

If you are familiar with other operating systems, you may expect local variables to be static. That is, on other systems, local storage is allocated before execution, initialized once, and not initialized again during the program run. Consequently, in these other systems, variables in local storage hold their values after a subprogram returns. If that subprogram is called again, at the start of execution local variables have the values they held at the time of the previous return. To convert programs written on these other systems to Multics, you should add a save statement or a %global or %options static statement. (See the Multics FORTRAN manual for a full description of these statements.)

Static storage is normally allocated in the free storage region and is initialized when the linkage section of the program is copied into the free storage region--in other words, when the program is referenced in your environment for the first time. Static storage is allocated for the life of a FORTRAN run. For run units (see "Run Units," below), the life of the run is until the run unit terminates. For all other program runs, the life of the run is until a new_proc or logout command is issued. Static storage is initialized only when it is allocated, and is not initialized again however many times the referencing program is called during a run. Static variables, once defined, remain defined.

When Large Array or Very Large Array static storage is used, storage for arrays is allocated in managed storage, with the pointers to the storage left in the static section within the free storage region. In such a case, that linkage section "owns" that managed storage. When the program's linkage section is released by deletion or termination, the managed storage is also released.

Local variables can be declared static with the save statement or the %global or %options static statement. Local variables not explicitly declared automatic, in the case where some variables are explicitly declared automatic with the automatic statement, are static by default. You should declare local variables static when their values must be saved from call to call.

Normal common storage is allocated at first reference or with the set_fortran_common command (see "Common Block References," above). Common variables are always static. When Very Large

Common (VLC) is used, it is allocated in the managed storage area and is initialized and allocated when the compilation is first called or when the first reference in a bound-unit is made to a non-VLC block with the same name as a VLC block bound in the same bound unit. In this case, the VLC is allocated in the managed storage area and initialized through the link-snapping mechanism.

AUTOMATIC VARIABLES

Most of you are familiar with the common and local static forms of storage in FORTRAN. As long as you restrict variables to these storage classes--declaring variables common or specifying them for local static storage with the save statement (or the %global or %options static statement)--you should experience no surprises. Automatic storage, however, is a concept alien to many FORTRAN users, and it can lead to a great deal of confusion because the compiler handles it in different ways. Program units compiled together reside in the same object segment. Automatic storage is allocated for the whole segment whether it contains one program unit or several. To understand the problems connected with automatic storage, you must first understand the significance of compiling many programs together from a single source segment.

PROGRAM UNITS COMPILED TOGETHER

If all the program units in a program are compiled together, you get something that resembles a load module: the system makes no distinction between the files that comprise the segment, and they all occupy contiguous storage. This has the added effect of increased efficiency of execution.

Quick Calls

The FORTRAN compiler attempts to create an object segment that executes as fast as possible. In so doing, it uses a very fast internal call/return protocol that takes advantage of the fact that FORTRAN is not a recursive language. In particular, when program units are compiled together, no separate stack frames are generated for the different program units, and thus stack frames are not created and destroyed in calls between such program units. This means automatic storage works differently when distinct program units are compiled together and when they are not. When distinct program units are compiled together in a single object segment, the automatic variables remain defined until control leaves the object segment. They are allocated storage and initialized only at the beginning of execution, and if a subprogram compiled with other subprograms is called repeatedly in the course of a run, the variables retain at each call the values they held at the previous return. So it goes until the subprogram returns to its caller for the last time.

When distinct program units are not compiled together, the values of automatic variables become undefined when a subprogram returns and are not retained for a later reentry to the subprogram.

The compiler sets up all entries to subprograms compiled together using the fast call/return protocol. The main significance of this fact is that automatic variables in subprograms so compiled are initialized only when the stack frame is created. Moreover, because the calling program and the subprogram it calls are in the same object segment, there is no new allocation of storage when the subprogram is called from within the segment. When the two program units are in different segments, by contrast, storage is allocated for all program units in the segment that contains the program unit being called. Thus, for the sake of programming efficiency, you should compile together programs that call each other, so as to minimize calls between segments. The treatment of automatic variables in subprograms compiled together is in direct contrast to the treatment of automatic variables in independently compiled subprograms. (In the case of subprograms compiled together, the designers of the compiler could have decided to have all automatic variables of a given subprogram initialized every time the subprogram is called--and the compiler may indeed do this in the future. At present, automatic variables in programs compiled together retain their values through successive calls and returns as long as the stack frame exists. The stack frame goes when the program returns to a caller outside the segment, such as an independently compiled program.)

Programs that work when compiled in a block may not work if broken apart and recompiled separately if they do not adhere to

the definition of automatic storage. (The ANSI Standard definition of automatic storage states that the values of automatic variables become undefined when a subprogram returns.) The same goes for programs that are first compiled separately and later compiled as a unit. Therefore, if you want to debug a particular program unit independently of other units in a program, you should compile it by itself first, and only after testing compile it in a block with the other units of the program. If the program is in full conformance with the ANSI Standard, it will work no matter how it is compiled.

Any subprogram or function that is passed to a routine as a parameter uses the long calling sequence rather than the quick calling sequence, whether it is within the same compilation unit or not. This causes a new stack frame to be laid down and all necessary initializations to be performed. The effect of such a calling sequence may noticeably degrade execution of the program. The only optimization possible for this situation is to compile separately all routines to be passed in such a manner into their own compilation unit, thus limiting the initializations performed. In addition, since a new stack frame is laid down, the automatic variables known within the calling program are not initialized within the called program, even if it is within the same compilation unit, unless they are formal parameters passed by the calling routine.

SEPARATELY COMPILED PROGRAM UNITS

When program units are compiled separately, automatic variables are allocated storage at the beginning of execution of each subprogram. When a subprogram returns to its caller, automatic variables are released and their values become undefined. Storage is allocated anew at each subsequent call, and ceases to exist at each return. Automatic variables exist, in this case, only for the execution time between call and return. These automatic variables are initialized to zero each time the program referencing them is invoked, unless the program contains a %global no_auto_zero statement or explicitly initializes variables with a data statement or an assignment statement. As a matter of good programming practice, it is best to initialize all variables in assignment statements, to ensure that they will behave the same way no matter how the program units are compiled. (See "Failure to Initialize Automatic Variables," below). Programs written to depend on the initialization to zero of automatic variables that have no specified initial values may not work on other systems if transferred from Multics. However, the most efficient way to initialize is in bulk at allocation time. If a program's variables are to be initialized, and performance is the issue, that is the time to do it. If portability is the issue, use static variables. (See "Undefined Variables," below, for some further factors to consider.)

FAILURE TO INITIALIZE AUTOMATIC VARIABLES

In the Multics system, an automatic variable whose value has become undefined after a return is initialized to zero the next time it is referenced, unless the program that references the variable explicitly initializes it to some other value (unless %global no_auto_zero is specified). Thus when an independently compiled subprogram is invoked, uninitialized automatic variables are set to zero, and their values become undefined after a return. Since the standard specifies that variables in local storage become undefined after a return, variables given zero values by Multics when storage is allocated for them are, strictly speaking, undefined throughout execution until defined in an executable statement. A program that references variables whose values are undefined is not a valid program. Programs consisting of separately compiled program units that reference automatic variables of undefined value may work in the Multics system, but to expect that they will is a programming mistake. Moreover, it is not recommended that you count on variables in independently compiled program units to be initialized to zero, primarily because the same group of program units will execute differently if they are later compiled together. In particular, it is a mistake to expect automatic variables in separately compiled program units to retain their values from call to call. A separately compiled program unit written in the expectation that local variables retain their values over a succession of calls is in error. The compiler will correct the error when program units are compiled together. You are encouraged to declare variables that are to be retained static by using the save statement (see below), and to explicitly initialize automatic variables in the subprograms in which they are referenced. (All Large Array and Very Large Array automatic storage is automatically zero at all times, since the segment is truncated when returned to the storage manager.)

Initialization

If a program relies on the compiler to initialize automatic variables to zero (or initializes automatic variables with a data statement) in the expectation that the variables will be initialized at every call, the program must not be compiled in one segment with any programs that call it. The reason is that automatic variables are not initialized every time a program unit is called within the segment, but only when the call originates "externally," that is, from a program unit in another segment. Another way of saying the same thing is that the behavior of the program will vary as a function of how the program was compiled. This situation is extremely undesirable, and you are encouraged to avoid it either by explicitly initializing all automatic variables by the use of assignment statements or by declaring variables static either with the save, %global save, or %options save statements or in common blocks.

UNDEFINED VARIABLES

All variables in automatic local storage become undefined when their storage is released. In implementations where storage is allocated only once for a program run, the values of local variables can be expected to stay around from call to call. In Multics, storage for program units compiled together is released after the last return, and storage for separately compiled program units is released each time a program unit returns. The values of variables, after release, are truly undefined. There is no way to control with certainty what values variables contain once they have become undefined.

It cannot be overemphasized that programs written to take advantage of the fact that, on other systems, local storage is so allocated that in fact variables do not become undefined may not run on Multics. Such programs are in error according to the ANSI Standard, which states that uninitialized variables in local storage become undefined after a return. If you add a generalized save statement to such programs, however, they will execute as expected. Such programs do not work as effectively with the optimizer as those that do not save variables globally, however. In the Multics system, automatic variables become undefined 1) when their storage is released; 2) when a program run terminates.

SAVE STATEMENT

In local storage, you can explicitly declare variables to be static with the save statement. A save statement in a subprogram without an accompanying list of variables causes all variables in the subprogram to be declared static. These static variables are allocated for the entirety of the program run, no matter how many times a subprogram is called and returns.

This page intentionally left blank.

If you wish to specify a number of variables for static storage, use a save statement. All variables not specified will have the automatic storage class attribute. If you use a save statement to specify a list of variables in a subprogram, only those variables are static and only in that subprogram; in this way, for example, you can declare a counter variable to index the number of times a subprogram is called in a particular run.

The save statement is standard in Fortran 77.

Example:

```
subroutine x
save c
c=c+1
return
end
```

AUTOMATIC STATEMENT

The automatic statement makes it possible for you to specify a set of variables for automatic storage. Its effect is to change the default to static; that is, variables not specified in the list of the automatic statement are declared for static storage. Variables declared automatic should be explicitly initialized in the subprograms in which they are referenced.

The automatic statement is an extension to standard FORTRAN. It is included in Multics FORTRAN for the sake of convenience; if you want to save most of a long list of variables, but wish to declare a few to be automatic, the automatic statement enables you to specify the automatic variables and thus to save all the others.

DATA STATEMENT

Automatic variables are initialized by the data statement whenever new storage is allocated. For subprograms compiled with a main program, storage remains allocated until the run terminates. For subprograms compiled separately, storage is allocated every time the subprogram is called. For subprograms compiled together with the main program, the data statement initializes automatic variables only once per run. After the first time such a subprogram is called, the data statement has no effect on the values of variables. A data statement in an independently compiled subprogram initializes automatic variables every time execution enters the subprogram. In other words, when a data statement is used to initialize variables in a separately compiled subprogram, the variables get initialized values at every call because the allocation is thrown away at every return.

In the case of programs compiled together, the use of a data statement to initialize automatic variables in subprograms is not recommended. Whether or not you compile your programs together, you should explicitly initialize automatic variables with an assignment statement in the subprogram in which they are referenced. In the case of an independently compiled program unit, that data statement is sufficient, but, again, if that program unit is later compiled with others, the effect of the data statement may change. It is therefore recommended that you use the assignment statement to initialize automatic variables unless there is some important reason not to do so.

The data statement is chiefly useful for the initialization of static variables, whether local or common (see "Common Storage" below). Local variables initialized with the data statement should be declared static with a save statement.

Run Units

As already mentioned, the inclusion of the RNT among the entities searched when the linker is looking for a reference name can lead to the linker's finding a segment other than the one you have in mind. There are three reasons why this unexpected and confusing error can occur. First, you may be unaware that the reference name has already been initiated for another entirely independent segment. Second, you may have forgotten that the name has already been initiated. Third, you may have several programs with the same name, in the expectation that the system will find the "right" one each time.

Nearly all these problems can be solved with the run command, which in essence sets up a new FORTRAN environment, called a run unit. The run unit carries with it none of the history of your environment before the run. (This history includes initiated reference names in the RNT, common block storage, and static storage.) The new environment brings with it a new free-storage region and a new, empty RNT. All storage allocated in the free-storage region is effectively reinitialized, including normal common blocks and storage declared static with the save statement.

Permanent common blocks, being segments in the Multics storage system at large, may not be the same before or during a run unit, depending on how the search rules for the two environments are set up. Automatic storage is always initialized when the stack frame containing the storage is allocated.

If the first program to reference a common block is not compiled or bound with the block data subprogram that initializes the common block, the common block may not be successfully initialized. The set_fortran_common command circumvents this possibility by initializing common storage for a FORTRAN run.

USE OF THE RUN COMMAND

The run command has a powerful but very simple user interface. In its simplest form, it takes a single argument--the name, including the entrypoint name, of the main program of the run. A new environment is established and the main program is called. On return from the main program--when execution terminates in an end line or a return or stop statement--the environment is cleaned up and discarded. The only lasting effects that FORTRAN programs can have if run in this environment, therefore, are in the use of input/output statements that change files or in the use of permanent common blocks. All automatic, local static, and normal common storage is discarded at the end of the program run. The prior environment is reestablished, with the result that the values in common and local static storage, and the reference names in the RNT, are backed up to their state just before invocation of the run command.

The initial RNT used by a run unit is empty unless the -copy_reference_names control argument of the run command is selected.

QUIT, START, RELEASE

Programs in execution or waiting for input can be interrupted with the Multics quit facility. You press the ATTN or BREAK key (whatever it is called on the terminal you use). The program is then suspended and a new command-level environment established. The old environment is not destroyed and can be restarted if desired via the start command. If you do not want to retain the old environment you can discard it and return to the original command level by use of the release command. If you issue the release command after a QUIT, unless the program is run within a run unit, the files used by the program are left in an inconsistent state. To clean up such files, use the close_files command. When the program runs within a run unit, files are automatically closed where appropriate.

Issuing a quit while running in the FAST or DFAST subsystems aborts the program completely, just as issuing a quit and then a release does in the normal Multics environment. All input/output files are closed and common blocks freed.

Due to the manner of implementation of fortran_io_ and its method of stack frame use, it may not be possible to restart programs in all instances, since it is possible to lose the stack frame information for fortran_io_ through unwinding of the stack.

PAUSE, START, RELEASE, STOP

If a pause statement is executed in a FORTRAN program in the FAST or DFAST subsystem, then the message "PAUSE NNN" is printed out (where NNN is the optional argument). If the pause statement is not executed in FAST or DFAST, then the fortran_pause condition is signalled and the info_string associated with the condition is set to the pause argument, if an optional argument exists. This condition is fully restartable and can be managed either through PL/I signal handlers or at command level, using the on command. If no signal handlers exist for the condition, the current program is suspended and a new command level established. At the new command level, you can issue normal commands. The two most relevant to the execution of the FORTRAN program are start and release. Issuing the start command after execution of a pause causes the FORTRAN program to resume execution of the statement immediately after the pause statement. Issuing a release command after execution of a pause statement causes the current FORTRAN program stack environment to be discarded and a return to the previous command level. Unless the program is run within a run unit, releasing the program in this way leaves the files used by the program in an inconsistent state. To clean up such files, use the close_file command. When running programs within a run unit, the files are automatically closed, if appropriate, whether the run is terminated normally, abnormally, or by the action of the release command. A stop statement in the program terminates execution and returns the program to its caller.

When the pause statement is executed in an absentee environment, the fortran_pause condition is signalled and the message "PAUSE NNN" is printed out (where NNN is the optional argument). If there is no condition handler established for the pause signal in absentee, then the program continues; a new command level is not established.

Figure 1-3 gives a pictorial view of what the stack segment might look like at different times during a FORTRAN program run: in Figure 1-3a, the last frame of the stack is for the command level programs. From command level, you can type commands at the terminal. Once a command is typed, for example the name of a FORTRAN program, that program is called and a stack frame immediately allocated for it (Figure 1-3b). The stack remains in this state for the duration of execution of the FORTRAN program.

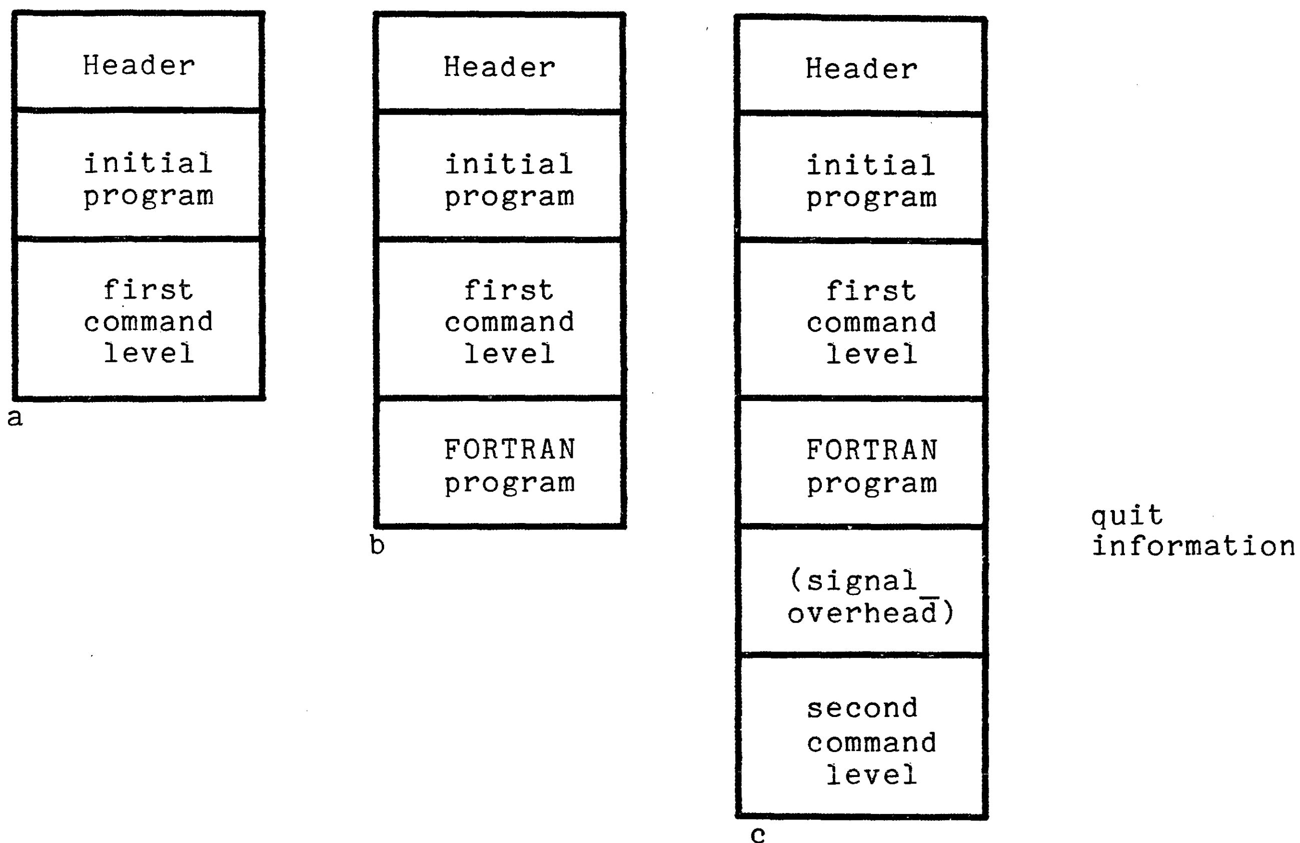


Figure 1-3. State of Stack

(a) State of Stack after Login
(b) State of Stack after Command is invoked
(c) State of Stack after Quit

Figure 1-3c depicts the stack after a quit is signalled (or equivalently, after a pause is executed). Here a second command level is established. The first command level, and the FORTRAN program itself, have been suspended but nothing has been thrown out.

At this point you may issue further commands. The start command would cause the FORTRAN program to resume execution, and the stack to revert to the state illustrated in Figure 1-3b. The release command would cause the stack frame (and hence the execution state) of the FORTRAN program to be discarded, and the stack to revert to the state depicted in 1-3a.

Note that it would be possible at the second command level (Figure 1-3c) to invoke the same FORTRAN program called at the first command level. This re-entrant execution of FORTRAN programs is ill-advised, since if the program references static local storage, common storage, or input/output statements there will probably be trouble. In fact, the files opened and used during the first invocation of the FORTRAN program would be used again in the second invocation. In the event that the first invocation is ever resumed, unexpected behavior is likely to be the result, since many variables, attributes, and so on have been changed during the second invocation of the program--unknown, of course, to the first invocation.

Automatic Storage in Stack Frames

Figure 1-4 illustrates several of the states of the stack during execution of a FORTRAN program consisting of several subprograms. The call/return sequence depicted is:

Program A calls program B
Program B calls program C
Program C returns to B
Program B calls program D
Program D returns to B
Program B returns to A
Program A returns to command processor

These diagrams illustrate the behavior of four separately compiled programs, each allocated a new stack frame each time it is called. (Recall that, in the absence of a %global no_auto_zero, this form of allocation initializes the automatic variables either to zero or to values specified in data statements.)

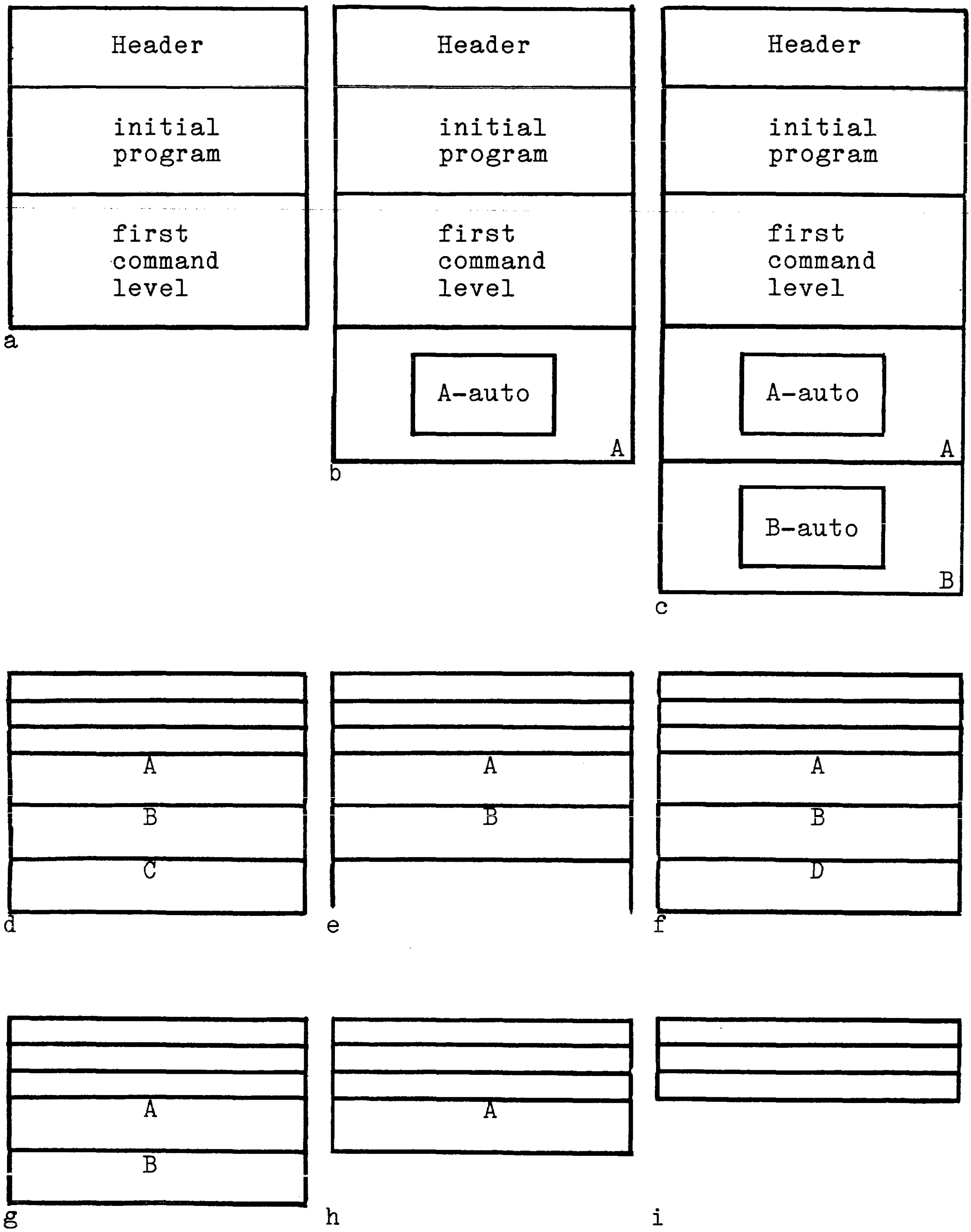


Figure 1-4. Allocation of Stack Frames

(a) User at command level

(b) A is invoked and gets stack frame, in which automatic variables are allocated and initialized.

(c) A calls B. B gets stack frame, in which automatic variables are allocated and initialized.

(d) B calls C, C gets stack frame, in which automatic variables are allocated and initialized.

(e) C returns to B, the stack frame for C is discarded, and storage is released.

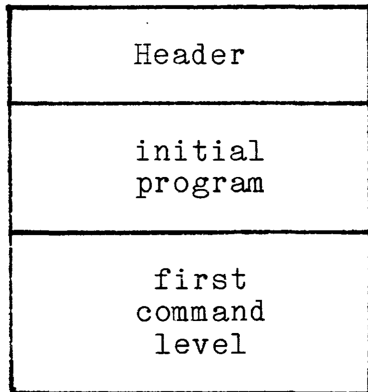
(f) B calls D, D gets stack frame, in which automatic variables are allocated and initialized.

(g) D returns to B, the stack frame for D is discarded, and storage is released.

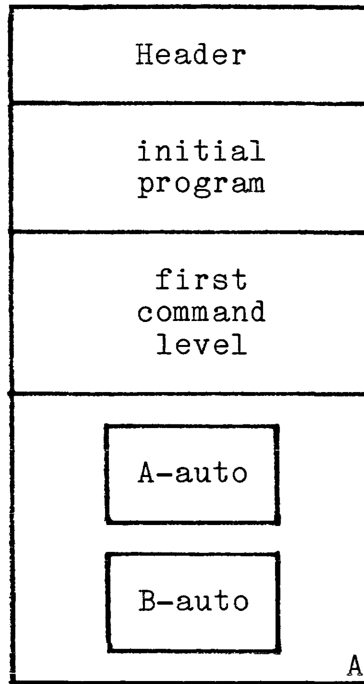
(h) B returns to A, the stack frame for B is discarded, and storage is released.

(i) A returns to command level. All program-specific automatic storage is released.

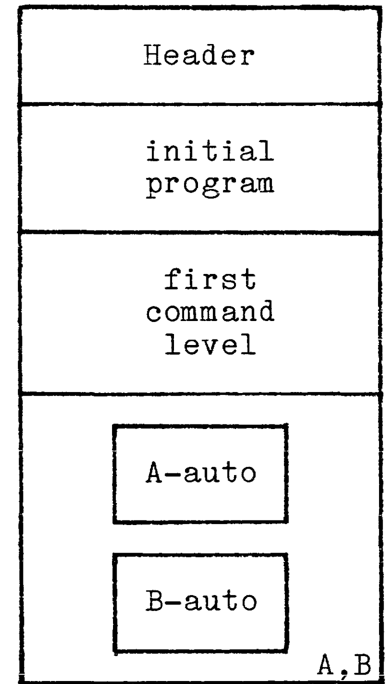
Figure 1-5 illustrates the states of the stack when the program calling sequence is the same as in Figure 1-4, but here programs A and B were compiled together and programs C and D were compiled together. Note that there is no change in the stack when programs compiled with other programs are called. (Since there is no change of state, the automatic variables are not initialized on these calls.)



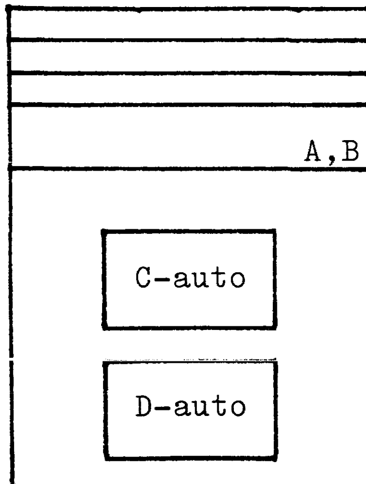
a



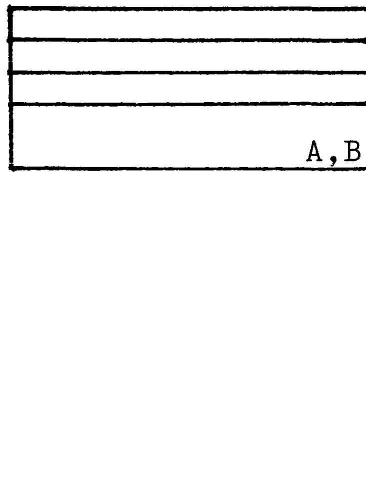
b



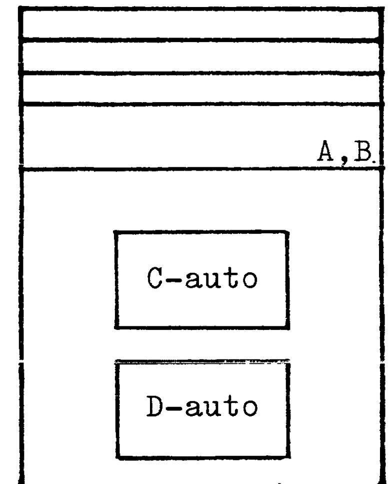
c



d



e



f

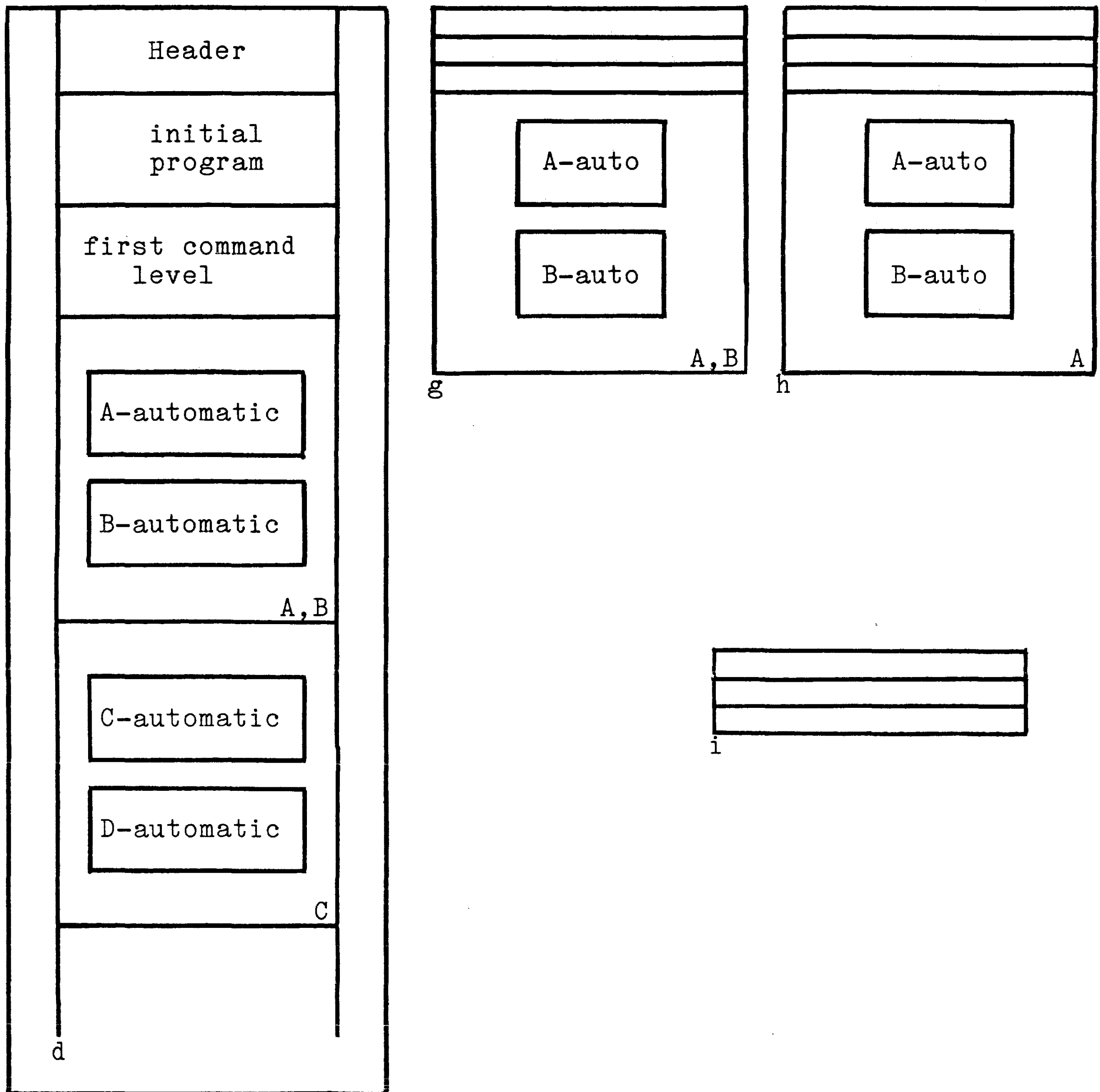


Figure 1-5. State of Stack in Program Compiling

(a) User at command level.
(b) A is invoked, and gets stack frame, in which automatic variables of both A and B are allocated and initialized.
(c) B is invoked. No new stack is created for it, because the automatic variables were allocated on previous call.
(d) B calls C, C gets new stack frame, variables for both C and D are allocated and initialized. Note that in state d, the automatic variables stored for D are never used. They are allocated anyway since the stack frame is the same.
(e) C returns to B, stack for C and D is released.
(f) B calls D, D gets stack frame back, variables for both C and D are allocated and initialized. Note that in state f, the call to D does not imply any usage of the variables stored for C, just as in state d.
(g) D returns to B. The stack frame for C and D is now released.
(h) B returns to A. There is no change in stack frame.
(i) A returns to command processor. The stack frame for A and B is now released.

Binding FORTRAN Programs

The Multics bind command can be used to merge several FORTRAN object segments into one. The end product is the nearest thing in Multics to a load module, and the binder is in some ways analogous to a link-edit program, since it actually resolves as many external references as it can in order to avoid the linkage faults that would otherwise occur at run time. Such references are resolved by the binder without recourse to the search rules--the binder looks only in the programs that are being bound, and rejects any programs in which there are ambiguous external references. Hence, many of the unexpected consequences (or side effects) of dynamic linking disappear when the binder is brought into play. This procedure offers the advantages of taking up less storage for the object code and avoiding many linkage faults that would otherwise occur if the bound programs referenced each other from separate segments. The object segments to be bound must not have been created using the -non_relocatable control argument.

The behavior of automatic variables is in no way changed by the binder. They are initialized according to the way they were originally compiled.

Because of the addressing methods used by most Multics object segments and the binder's knowledge of these techniques, it is impossible for the binder to bind together programs with local static storage in excess of 16,384 words. Hence, bound FORTRAN programs are limited to this amount of local static

storage. If there is a need for more storage, you can use automatic variables where appropriate, or common blocks. If you initialize a common block in a block data subprogram, the block data subprogram should be bound with subprograms that reference the common block, and, in particular, with the subprogram that first references it. A template of initial values for each initialized common block goes into the bound segment that contains the block data statement; hence it may be that a large initialized common block cannot be bound.

SECTION 2

ENTERING YOUR FORTRAN SOURCE PROGRAM

LOGIN

To log into the Multics system, type the command:

login <person>

The system types back a prompt for your password:

Password:

Then you type your password, which is either masked by characters output to your terminal, or simply suppressed. When you receive a ready message, you are logged in.

CREATING A SOURCE SEGMENT

Once you have successfully logged into the system, a ready message prints out on the terminal. You are now at command level, and the system is ready to accept commands. To create a source segment containing the ASCII text that represents a FORTRAN program, you must invoke an editor.

Those unfamiliar with time-sharing systems, and Multics in particular, may not have experience with an online text editor. A simplified introduction to the Multics text editors is found in the New Users' Introduction to Multics. The essential information for creating a FORTRAN source segment is provided below.

To create a FORTRAN source named test.fortran that prints a simple message, you could invoke the qedx text editor as follows (exclamation points indicate lines you type):

```
!  qedx
!  a
!        write(0,1)
!  1     format("What is the output of this program?")
!        end
!  \f
!  w test.fortran
!  q
```

Typing the Multics command qedx invokes the editor; the append request (a) signifies that the lines to follow are input to the editor; on succeeding lines the actual text of the program follows. (Since the Multics system differentiates between upper- and lower-case alphabetic characters, examples of FORTRAN text in this manual will be, for the most part, in lower case.) At the end of input, type \f, which terminates input and makes it possible to issue further requests to the editor. The write request (w) with the relative pathname test.fortran saves the input in a segment. A quit request (q) returns you to command level. A source segment, named test.fortran, is now created and listed in your working directory. The suffix ".fortran" is required.

In addition to these simple editor requests, qedx supplies means of altering and replacing character strings line by line, changing lines, deleting and inserting text. A basic subset of editor requests, as described in the New Users' Introduction to Multics, is a necessity for effective interactive use of the Multics system.

At this point, you may wish to edit or to compile your program. To edit, type qedx and read the segment into the editor using the read request (r) (see the MPM Commands manual for a complete description of qedx requests). To compile, see Section 3 of this manual.

INPUT FORMAT

Two formats can be used for source-program text: freeform and card image. In freeform format, each line is interpreted as a sequence of characters without consideration of column fields. Freeform format is a nonstandard option in Multics FORTRAN that makes it difficult to transfer a program to another system. Conversion of card-image programs to freeform programs is not recommended if the program is to be moved from Multics to another system. In card-image format, each line is interpreted as fields of characters that are defined by column positions.

Uppercase and Lowercase Letters

The FORTRAN compiler distinguishes between uppercase and lowercase characters (in particular, the compiler expects all keywords of the FORTRAN language to be lowercase), but since many other computer systems use only uppercase characters, an option is provided by which you can instruct the compiler to treat all characters (except those in character-string constants) as if they were lowercase. This option, specified at command level with the -fold control argument of the FORTRAN command (see Section 3), is useful in compiling programs brought over to Multics from other systems. It also makes it possible to write new programs that are intended to be portable to other systems entirely in uppercase characters. The -fold control argument is assumed when the -card control argument is specified (i.e., for card-image files); when the -fold control argument alone is specified, the uppercase source program is assumed to be in free-form format (see "Free-Form Format" and "Card-image Format," below. For a complete list of control arguments available with the compiler, see Section 3 of this manual.)

NAMES IN THE FORTRAN PROGRAM

Symbolic names in Multics FORTRAN programs may be from 1 to 256 characters long. Use of the underscore character (_) in such names is a Multics system convention and makes the program nonportable and nonstandard; so does the use of more than 6 characters. The dollar sign character ($) should appear only in names that reference external objects; the dollar sign character has a special meaning in Multics, and only one dollar sign character is permitted in a name (it cannot be the first character). (For information about dollar-sign names, see Section 1 of this manual.) Use of the dollar sign in a name within a program makes the program nonportable and nonstandard. External names must have no more than 32 characters before the dollar sign.

Free-Form Format

There are special conventions in free-form format for comment and continuation lines, the use of semicolons, and line numbers.

COMMENTS

If the first character on a line is an uppercase or lowercase c, text on that line is interpreted as a comment. An asterisk (*) or exclamation mark (!) that appears as the first nonblank character on a line can also be used to signify a comment. The appropriate use of the * and ! characters can enhance the readability of a listing: for example, comments can be grouped toward the right margin to distinguish them from code.

The effect of marginal notation can be further enhanced with the ! character, which is interpreted as the beginning of a comment wherever it occurs outside a character string constant. All text on the containing line after an exclamation mark is interpreted as a comment and is ignored by the compiler except for listing purposes.

Example:

```
475   p = p+1

c the first subscript in the array is assigned
c a value, since we have a prime number
     primes(p)=iprime
```

Example:

```
data primes(1),primes(2)/2,3/ !First two elements
                              !of array are initialized
                              !with data statement
```

Example:

```
data primes(1), primes(2)/2,3/

*
* data statement initializes first two array elements
*
```

Note that in free-form format any line with c as the first character is interpreted as a comment:

```
integer i
complex c
c = 0
print, "type the number"
input, i
c=cmplx(0.0,float(i))
i=int(aimag(c))
print,i
end
```

This example, in which all lines begin in column 1, will fail to compile because it contains three lines that the compiler interprets as comments. To be certain of obtaining the desired results in free-form format, you should make a practice of reserving column 1 for comments. To make the source program more readable, as well as to avoid the ambiguity exemplified here, it is recommended that all lines other than comments and those

containing statement labels be typed with an initial TAB character.

CONTINUATION LINES

If the first nonblank character of a line is an ampersand character (&), subsequent text is interpreted as a continuation of the previous line and concatenated to the text on the preceding line. There is no limit to the number of continuation lines in a FORTRAN source text (although there is a limit to the actual size of the statement. See Section 4 of this manual, "Constraints").

Example:

```
      y=array(k)*vect(k)/array(k+1)-0.7/array(k+1)
     &        +array(k+1)*array(k-1)*x
```

Note that an initial ampersand in column 6 can be used interchangeably as a continuation indicator in free-form or card-image format.

SEMICOLON

The semicolon character (;) can be used to separate statements on a single line, with the restriction that statements after the semicolon and on the same line cannot be labeled. (An end line must not contain a semicolon.) In general, the use of semicolons may make a listing difficult to read, and is sometimes regarded as a poor programming practice.

Examples:

Recommended

```
if(j.ge.i) goto 300
if(mod(i,j) .eg. 0) goto 100
goto 200
```

Not Recommended

```
if(j.ge.i)goto 300;if(mod(i,j).eq.0)goto100;goto 200
```

This sequence of statements is logically connected in a way that might tempt you to type them all on one line. Yet the result is that the actual flow of control is obscured.

If a source program contains an end followed by a semicolon, as

```
end;
```

the compiler will return an error message indicating that there is no end line in the program and that one is being supplied. If an end is followed by a semicolon and some other statement, the program might compile, but fail to run properly.

LINE NUMBERS

If you specify the -line_numbers control argument (see Section 3 of this manual), the compiler expects a line number at the beginning of each line. A line number is an unsigned integer of five or fewer digits, the highest permitted being 16383. The line number is terminated by the first nonnumeric character on the line (which may be a blank). Programs written in the FAST environment automatically have line numbers in the source segment. A compilation listing (see "Listing" in Section 3 of this manual) produces a line-numbered segment when the program is compiled. The line numbers in this case are not on the source segment before compilation. Each separately compiled program unit that has line numbers as input to the compiler is restricted to 16383 lines. Programs that do not have line numbers are not so restricted, but the probe and debug commands will fail to work on compilation units longer than 16383 lines. (The term compilation unit refers to any group of one or more program units compiled in one invocation of the compiler.) If the entire program is more than 16383 lines long, it might be convenient for debugging purposes to compile the individual units separately, and compile or bind them together afterward. (See the MPM Commands for a description of the bind command, or "Bindign FORTRAN Programs" in Section 1 of this manual.)

Card-Image Format

Source segments originally in the form of card decks can be compiled in card-image format using the -card control argument. With the -card control argument, the -fold control argument is assumed, and uppercase and lowercase characters in the source are not distinguished except in character-string constants. In card-image format, every source line is treated by the compiler as being 80 characters long, of which the first 5 characters (except for comments) either are blank or contain a statement label. The sixth character either is a blank or indicates a continuation. Lines shorter than 72 characters are padded on the right by the compiler with enough blanks to fill out the line to 72 characters. Characters in columns 73 through 80 are ignored. A correct FORTRAN source segment moved to Multics from systems using card decks may not compile in free-form format, but will usually compile correctly if you use the -card control argument. Conversion from card-image format to free-form format requires that all Hollerith and character-string constants affected by the format change be examined, since if a Hollerith field or a quoted character string of blanks continues to a new line, the blanks may disappear if you do not ensure that they are retained.

The following program gives an example of a character string in a Hollerith field that is continued:

```
      write(6,10)
10    format(56h
     &,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x
     &,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x)
      stop
      end
```

If you compile this program as a free-format program -- that is, without the -card control argument -- its output looks like this:

```
,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x
```

If you compile the same program with -card, it will produce blank lines as output. The point is that in card-image format, statements get paded on the right out to column 72. Thus in card-image format, the 56h eats up the first 56 characters of the ",x,x,x... " string. Further, the use of the horizontal tab character in card-image source is strongly discouraged. The tab is a single column. Multics FORTRAN considers a tab to occupy a single column, even though on the terminal it may appear to use up more column depending on tab settings and where on the line the tab appears.

Card-image format differs from free-form format in certain details with respect to comments, continuation, and line numbers.

COMMENTS

If the first character in a line is c, C, or an asterisk (*), the line is interpreted as a comment. The actual comment can be typed anywhere on a line after an initial comment character. The "*" character is a nonstandard comment indicator.

CONTINUATION LINES

If the sixth character on a line is nonblank and nonzero the line is interpreted as a continuation line.

LINE NUMBERS

Line numbers are not an option in card-image format.

SECTION 3

COMPILING AND EXECUTING THE FORTRAN PROGRAM

Invoking the Compiler

The FORTRAN compiler is invoked by the fortran command. The only argument required by the fortran command is the name of a FORTRAN source segment. The name of the source segment must end with the .fortran suffix.

```
source.fortran
```

Hence, to compile a source program named test.fortran, the command line

```
fortran test
```

is sufficient. The compiler automatically adds the .fortran suffix, if it is not specified in the command line, before trying to access the source segment. If you give only the entryname, the source segment is assumed to be in your working directory, and the object segment that is created will be listed there.

The example above illustrates the simplest kind of compilation. When, as above, you select no optional features, the only output the compiler produces is the executable object segment itself and messages describing any errors the compiler detects. The compiler itself never tries to execute the FORTRAN program.

The object segment is created and catalogued in your working directory under the name test, i.e., the name given as an argument to the compiler. If you were to list out the contents of your working directory, you would now find in it the entrynames:

```
test
test.fortran
```

A program may or may not compile successfully. If it fails to compile, the compiler prints error messages that describe errors in the source program, and notifies you that translation has failed. For example:

```
ERROR 116, SEVERITY 3 ON LINE 5
Syntax error. Decimal point missing from end of logical constant.

ERROR 104, SEVERITY 2 ON LINE 19
Missing end line. One will be supplied by the compiler.
new_fortran: translation failed. test.fortran
```

The compiler returns, and you are again at command level, where a ready message is printed on the terminal. To correct the source program, invoke one of the Multics text editors. This edit/compile cycle typically continues until there is an error-free compilation. (For more information about object segments see Section 1 of this manual. For a complete list of error messages see Appendix A of this manual.)

ERROR DIAGNOSTICS

Since in the early stages of the development of a program there may be many errors, even when a program compiles, you have the option of identifying errors in the source before the compiler creates an object segment.

By compiling the program with the -check control argument, you request the compiler to diagnose errors in the source, and print error messages, without creating an object segment. The time and expense of creating an object segment that cannot be executed successfully are thereby avoided, while you have the opportunity to identify and correct errors that can be corrected without consideration of the object segment. Note that the -check control argument is intended for use when the source segment might reasonably be expected to cause some object code to be generated, even if not a complete object segment. Thus it saves you the expense of code generation until you have removed errors the compiler can detect.

The compiler performs the syntax checks and produces error messages whether or not you select the -check control argument. The -check control argument, in other words, only prevents the creation of an object segment. Also, programs compiled with the -check control argument must be compiled again without it before execution is possible. There is a class of semantic errors that are not checked if an object segment is not created. Therefore it is possible to compile a program with the -check control argument and receive no diagnostics, only to have new errors reported when the program is compiled without it. Of course, the program may simply not work. There are errors that can never be detected by the compiler or the FORTRAN runtime I/O routines; these errors will manifest themselves during execution of the program.

The following control arguments specify the options available with the FORTRAN compiler:

-ansi66
  compiles the program using the 1966 ANSI standard interpretation of incompatible constructs.

-ansi77
  compiles the program using the FORTRAN 77 standard interpretation of incompatible constructs.

-auto
  makes automatic the default storage class. Ignored if a save or automatic statement is used in the program.

-auto_zero
  initializes to zero automatic storage for local variables when they are allocated.

-no_auto_zero
  leaves initial values of variables undefined.

-brief, -bf
  prints error messages in short form.

-brief_table, -bftb
  generates statement table for debugging.

-card
  specifies card-image format.

-check, -ck
  performs only a syntax check of FORTRAN program (no object segment is created).

-check_multiply
  checks for single-precision overflows in integer multiplications. Conflicts with -no_check_multiply. This is the default unless optimization is requested.

-no_check, -nck
  produces an object segment.

-no_check_multiply
  inhibits checking for single-precision overflows in integer multiplications.

-default_full, -dff
  sets the default optimizer to "full_optimize" (see also -optimize). (Default)

-default_safe, -dfs
  sets the default optimizer to "safe_optimize" (see also -optimize).

-fold
  maps uppercase to lowercase.

-no_fold
  uppercase letters are not mapped into lowercase form.

-free
  specifies that the source segment is in free form format.

-full_optimize, -full_ot
  invokes the full optimizer to speedup program execution and reduce its size.

-large_array
  specifies that the compiler is to take all arrays in static and automatic and collect them for Large Array processing.

-no_large_array, nla
  specifies that large array support is not needed. (Default)

-line_numbers, -ln
  expects line numbers on input source.

-no_line_numbers, -nln
  indicates that the source segment does not contain line numbers.

-list, -ls
  produces source listing, including assembly listing.

-long_profile
  enables actual timing of execution speed according to the system realtime clock and does actual page fault counts.

-map
  produces source listing without assembly listing.

-no_map
  specifies that no program listing is to be produced.

-non_relocatable, -nrlc
  makes relocation impossible.

-optimize, -ot
  invokes the default optimizer (or the full_optimizer) to reduce object code size and increase execution speed.

-no_optimize, -not
  specifies that optimizations are not to be performed.

-profile, -pf
  permits metering execution of statements.

-relocatable, -rlc
  generates relocation bits for use by the binder.

-round
  uses rounded arithmetic for real and double-precision computations.

-safe_optimize, -safe_ot
  performs like -full_optimize, except that some code movement is inhibited.

-severityN, -svN
  suppresses error messages at terminal.

-static
  makes static the default storage class for variables in the program. Ignored if a save or automatic statement is used in the program.

-stringrange, -strg
  produces range checking code for all substring references.

-no_stringrange, -nstrg
  specifies that substrings are not to be checked to see if they lie entirely within string bounds.

-subscriptrange, -subrg
  checks for subscript values exceeding declared dimension.

-no_subscriptrange, -no_subrg
  specifies that subscripts are not to be checked to see if they lie within array bounds.

-table, -tb
  generates full symbol table for debugging. This is the default unless optimization is requested.

-no_table
  inhibits generation of full symbol table.

-time, -tm
  prints table giving compilation time.

-time_ot
  prints out timing information on the sub-phases of the optimizer.

-truncate
  use truncated arithmetic for real and double-precision computations.

-version
  prints out the version of the FORTRAN compiler before compiling.

-no_version
  specifies that version of the current compiler is not to be printed out.

-very_large_array, -vla
  specifies that the size of individual arrays may exceed a segment in length.

-no_very_large_array, -nvla
  specifies that very large array support is not needed.

-vla_parm
  specifies that the size of parameters passed may exceed a segment in length and that vla addressing must be used for parameters.

-no_vla_parm
  specifies that the size of parameters passed may not exceed a segment in length and that vla addressing must not be used for parameters. (Default)

Figure 3-1. Control Arguments

CONTROL OF ERROR MESSAGES

All warnings and errors output by the compiler are grouped into five classes or levels of severity as follows:

1 Warning only. Compilation continues without ill effect.

2 Correctable error. The compiler attempts to remedy the situation and continues, possibly without ill effect. (As in the example above, where the compiler supplies a missing end line. Correction of the source segment by the compiler does not guarantee correct results, however.)

3 An uncorrectable but recoverable error. That is, the program is in error and cannot be corrected during compilation, but the compiler continues processing up to the point just before the creation of an object segment. Any further errors in the source are diagnosed. If the error is detected during code generation, code generation is completed although the resulting object segment is

incorrect and is not executable. After compilation, a message is printed on the terminal informing you that an error of severity 3 has occurred.

4 An unrecoverable error. The compiler cannot continue beyond this point, and compilation aborts. Such an error is usually a result of implementation limits or errors in the compiler itself.

5 An unrecoverable error. Invalid control arguments are specified or the source program is not found. The compiler is not invoked.

You can set the "severity level" with the -severityN control argument so as not to be bothered by minor error messages; it is up to you to determine what severity level you wish to ignore. (The -severityN control argument affects only the compilation it is specified for.) If the severity level is set to 4, for example, all error messages are suppressed except severity 5 errors. Errors of severity 5 cannot be suppressed. Use of the -severityN control argument during the debugging phases of program development is not recommended, since these error messages can provide help in uncovering bugs. If, during the course of an editing/debugging session, you suppress error messages below a certain level of severity, new error messages below that level will not be printed on your terminal, and you would not know about any such errors that crop up as a result of debugging changes. As an alternative, you can specify the -brief control argument in the event that you want to see the messages in some form but also want to save time at the terminal. The -brief control argument causes the compiler to print a shortened form of all error messages. In the example above, the error message whose long form was:

```
ERROR 116, SEVERITY 3 ON LINE 5
Syntax error.  A binary operator is required in place of
               the name goto 30
```

would appear in its shortened form as

```
ERROR 116, SEVERITY 3 ON LINE 5
```

Once an error message appears on your terminal in long form, all further instances of that error message for a single compilation appear in a shortened form, whether or not you specify the -brief control argument. The -brief control argument is probably more useful than the -severityN control argument.

LANGUAGE OPTIONS

Multics FORTRAN is being brought into conformance with the 1977 ANSI standard for FORTRAN (FORTRAN 77). As this process is carried out, certain incompatible changes to the language must be introduced. To reduce the impact of these changes, two options are available for controlling the interpretation of constructs whose meanings are different under FORTRAN 77.

Under the ansi66 option, the "old" interpretation of incompatible constructs is used. The interpretation corresponds to the 1966 ANSI standard for FORTRAN with many extensions specific to Multics FORTRAN.

LISTING SEGMENT

Use the -map or -list control arguments when you need a listing of the program. Both control arguments produce a complete line-numbered source program listing, and the -list control argument produces an assembly-like listing of the compiled program. Use of the -list control argument significantly increases compilation time (and the cost of your listing) and should be avoided whenever possible with the -map control argument. The -map control argument produces enough information to allow you to debug most problems online. It creates a listing segment giving the correspondences between line numbers and object locations, the correspondences between names and octal object locations, and a list of statements and object locations. It does not provide an assembly-like listing of the object code itself. Both types of listing will appear in a segment with the .list suffix

```
test.list
```

in the working directory. This segment can be printed online, on the high-speed printer via the dprint command, or examined with a text editor.

Format of listing Segment

The listing segment created by the fortran command invoked with the -map or -list control argument begins with header lines specifying the absolute pathname of the source segment, the version of the FORTRAN compiler used, the date and time of compilation, the control arguments requested, and the options specified by %global statements. This information is particularly useful in the event that a bug turns up in a program that has been in use over a period of time.

After the header line, the following information is provided for each program unit in the compilation:

- A line-numbered ASCII listing of the source segment of each program unit. The compiler provides line numbers if you do not.

- An alphabetical table of all names, except statement labels, used in each program unit. Each name appears with its attributes, such as mode, storage class, and location, and a list of all lines on which it is used. If the code generator is not invoked, that is, the compiler is invoked only with the -check control argument, only names, except statement labels, appear in this table.

- An alphabetical table of all names, except statement labels, declared in the program but not used. If the name is not a member of a common block, that name is not allocated storage. Each name appears with its attributes and the line on which it is declared.

- A table, in ascending numeric order, of all statement labels in the program unit. Each label appears with the type of statement with which it is associated, its location if it is associated with an executable statement, the line on which it is declared, and a list of lines on which it is used.

- A table associating each executable source line with an object location. The table is arranged by ascending line number. It is available only if the code generator is invoked.

- A list of error messages for the program unit. All error messages appear in their long form.

- An assembly-like listing of the object segment. Each executable statement appears followed by the executable instructions generated for that statement. Each instruction appears on a line with the octal representation of the instruction word, and an assembly-like representation of the word including operation-code, pointer-register, and modifier mnemonics. All offsets in the assembly representation are decimal numbers. If the address field of the instruction uses the IC (self-relative) modifier, the absolute text locations corresponding to the relative address is printed in the remarks field of the line. If the reference is to a constant or name you have declared, the name is printed in the remarks field of the line. The assembly-like listing is provided only if you specify the -list control argument.

- An assembly-like listing of all constants allocated in the object segment. This is provided only if you specify the -list control argument.

- A table showing the storage requirements for the object segment. This table gives the size and offset for each section of the object segment. The size of the stack frame of the object segment is also given.

- An alphabetized list of all entrypoint names defined in the compilation. Each entrypoint name appears with the object segment offset for its external entrypoint, the program unit in which it appears if it is not a main entry point, the line on which it appears, and all external reference names in other program units that are resolved by it.

reference names in other program units that are resolved by it.

- An alphabetized list of all external references made that are not resolved within this compilation, including the lines on which they are referenced.

- An alphabetized list of all common blocks, the lines on which they are declared, and the declared length for each declaration.

- A list of all source segments used in the compilation, including the source program specified in the fortran command line, as well as any source segments referenced by %include statements. (See the Multics FORTRAN manual for a description of the %include statement.)

OPTIMIZATION

Use of the optimizer tends to reduce the execution time and size of the program, while increasing compilation time. The -optimize control argument should be used only after a program is fully debugged. The -optimize control argument performs the following global optimizations: removal of common subexpressions, removal of invariant expressions from loops, strength reduction, test replacement, constant propagations, and removal of assignments made dead by other optimizations. (See Appendix B of this manual for further information on these and other optimizations.)

The -safe_optimize control argument prevents some code from being taken out of loops by the optimizer. All other optimizations implied by the -optimize control argument are performed. Invariant operations that are not always executed on entry into a loop are not removed if it is possible that these operations could cause the fixedoverflow, underflow, or overflow conditions to be signalled. Assignments and operations that could cause the zerodivide or error conditions to be signalled are never removed from the above-mentioned portions of a loop, whether or not the -safe_optimize control argument is specified. (See Appendix B of this manual for information about the optimizer.)

The -optimize control argument will give correct results for most programs compiled without the -safe_optimize control argument. Only under the following--and very unusual--circumstances will the -safe_optimize control argument be necessary: if the -optimize control argument causes a valid program to signal the fixedoverflow, underflow, or overflow conditions, when such conditions were not signalled for a nonoptimized program.

Optimization requires some additional room in the operand region for the converter to store certain temporaries it creates. Thus, some large programs will not compile with the -optimize control argument. To circumvent this difficulty, the program may be divided and placed into two segments. To maintain the speed of calls under this circumstance, as many of the subroutines as possible should be made local.

The -optimize control argument should not be used with the -table control argument, since the optimizer may remove information that would otherwise appear in the symbol table. Debug programs before optimization.

IMPROVING PROGRAM SPEED

The FORTRAN compiler supports two types of program profiling. The -profile control argument enables you to generate additional code that meters the number of times each individual statement is executed. The -long_profile control argument enables actual timing of execution speed according to the system realtime clock and does actual page fault counts. The -long_profile control argument produces a more accurate representation of execution speed and overhead, but a program compiled with -long_profile takes considerably longer to execute than one compiled with -profile due to the overhead of actual measurement at runtime, as opposed to time estimation at profiling time. The -long_profile control argument is also capable of showing IO overhead and the time taken by called routines, which -profile is unable to do.

After you have developed a program that compiles and executes correctly, it may be desirable to speed the program up. The profiling arguments enable you to determine what parts of the program, and the associated subprograms, take up the greatest amount of execution time. After execution of a program, use the profile command to print the execution count (see the description in the Commands and Active Functions manual). Statement costs shown for optimized programs may be misleading, in that code may be moved by the optimizer from one statement to another, and be charged to the statement it is moved to rather than being charged to the statement for which the code was created. See Appendix B of this manual for more details about optimization.

CARD-IMAGE AND FREE-FORM SOURCE PROGRAMS

The -card control argument is used for compiling source programs written according to either the ANSI66 or ANSI77 standard; that is, source programs that are not freeform. It specifies that the source program is in card-image format, and it implies the -fold control argument, which maps all uppercase letters not occurring within character-string constants to their lowercase form. The -fold control argument alone is also useful for compiling source programs that are input either partly or wholly in uppercase characters. Programs with FORTRAN keywords in uppercase and variables in lowercase, for example, will not execute correctly in Multics unless compiled with the -fold control argument. If you need to modify an existing program by adding new material, such as a subroutine, in lowercase (typing it in from the terminal), the program as a whole will not execute correctly if the older stratum of the code is in uppercase characters and the modifications are in lowercase. The -fold control argument is the only way to accommodate situations like those just described, short of typing the whole source over in lowercase, since the -fold control argument distinguishes character-string constants from the rest of the code. The -fold control argument is particularly useful, in short, for compiling uppercase source programs as free-form format segments, as if they were typed consistently in lowercase characters.

DEBUGGING

There are two Multics system commands that are used in debugging: the probe command and the debug command. (For more information about debugging, see Appendix A of this manual, in which debugging is described in detail; and the description in the MPM Commands manual of the probe and debug commands.)

The -table control argument generates a full symbol table giving the correspondences between source line numbers and object locations, and a table with the name and location of each variable. The presence of such information in the object segment makes it possible to ask the debugger for variables by name. The -table control argument is useful with the probe command and with the debug command. Note, however, that the -table control argument adds significantly to the size of the object segment created.

The -brief_table control argument generates a partial symbol table that gives the correspondences between line numbers and object locations. Perhaps the most important function of the -brief_table control argument is that it permits runtime error messages to get the line number where the error occurs. Also, with either probe or debug, you can set "breakpoints" at different lines of the program. A program compiled with the -brief_table control argument could also be debugged with the debug command, but not with the probe command since with the debug command it is possible to locate variables by octal address (such information is available in the listing segment, if one exists). It should be emphasized, however, that the debug command is useful chiefly for machine-oriented debugging, and that the probe command is useful chiefly for source-oriented debugging. For debugging FORTRAN programs, the probe command is recommended. The -brief_table control argument is not useful for debugging with the probe command because probe does not use machine-level symbolic information.

In general, it is desirable to have a listing segment online for a program that is to be debugged. If storage considerations make this impossible, a hard copy of the source listing will serve as well. If there is no such hard copy, debugging can still be accomplished using a hard copy of the source program alone if the program was originally compiled with the -table control argument. If the program was not compiled with this control argument, recompilation may be necessary, with modification of the object code a possible result. Debugging--even with probe--in the absence of some form of source listing is not recommended practice. Although the probe command itself looks at the program online, it is in general a good idea to have a printed listing ready at hand for quick reference

during interactive debugging sessions at the terminal. (See Appendix A for a more detailed discussion of these and other debugging issues.)

The -table control argument may not produce the results you expect if the -optimize control argument is specified in the same compilation, since the optimizer may remove some of the information that would otherwise appear in the full symbol table. In the development stage, a program should be debugged thoroughly before you compile it with the -optimize control argument. (See "Improving Program Speed" and "Optimization," above.)

Note that the -list and -map control arguments (see "Listing Segment," above) have no effect on the object code, and that in order to use the debug and probe commands in debugging, the -brief_table or -table control arguments must be specified in order to make the information presented in the listing available to the debuggers.

Executing a FORTRAN Program

FORTRAN programs that are executed in the standard Multics environment can take advantage of (and are governed by) the powerful conventions of the Multics system. A FORTRAN program can be called directly from command level, like any command, merely by giving its name to the command processor. (The language at present does not easily handle arguments.) For example, if test_prime is a program that reads a number from the terminal and determines whether it is prime, the program would be invoked from command level as follows:

```
!  test_prime

   Input number to test:

!  4617

   The number is not prime.

   r 14:16 0.137 3.142 18
```

If the program accesses storage system files or external storage devices, some advance preparation prior to execution may be needed (see Section 5, Input/Output in Multics FORTRAN), but the name of the program is always the means of executing it.

SECTION 4

CONSTRAINTS

The following paragraphs give the limitations imposed in Multics FORTRAN on the size of records, files, programs, statements, arrays, common blocks, bound segments, and stack segments.

LENGTH AND FORM OF RECORDS

The length and form of a formatted record is determined by the format specification and the output data transfer list used to create the record. The number of records input by a formatted read statement is a function of the number of list elements and of the content of the format specification. The formatted read may be either sequential or direct access. The number of records output by a formatted write statement is a function of the number of list elements and of the format specification.

On input, formatted records are padded on the right with as many blanks as are required by the specified input format. A record delimiter (slash) in the format statement causes a new record to be read when it is processed.

The total number of computer words represented by the items in an unformatted input list must not exceed the total number of words represented by the unformatted output list that originally created the record. The elements in both lists should be of the same mode. The data modes must match; if you use mixed modes in unformatted data transfer lists your program is in error and you must be aware of the internal representations of the items involved, as well as the difficulties that stem from such use of mixed modes.

Unformatted read and write statements read or write a single unformatted record to or from the file. The maximum record length allowed for an unformatted record is close to the size of a segment, allowing for the overhead requirements imposed by the associated I/O module.

Files

In general, files contain records of varying length. There is no default maximum record length. A maximum record length can be attributed to a vfile_ blocked file (see Section 5 for a description of attributes). In such a case all the records allocated are of the same size, as far as the storage they occupy, although the record may not take up all the storage available for it. You must take care to ensure that the records in an existing file for which you specify a maximum record length attribute are not longer than the value specified. A write statement that creates a record longer than the specified maximum is in error.

IO Transfer Limits

Multics FORTRAN IO works through Multics IO DIM (Device Interface Module), which actually performs the IO operations. To this date, all DIMs are capable of transferring a segment or less in a single operation. This limits the size of a binary IO operation, which is record oriented, to a segment or less. Thus, you may not be able to do binary IO of arrays, or combinations of arrays that exceed this single-operation size limit. This will be particularly noticeable when you are using Very Large Arrays.

Programs

The maximum segment size is 261,120 words of storage. An object segment (the compiled version of a main program and all the associated subprograms) may not exceed this length.

Statements and Line Numbers

The maximum size of a single statement is 1320 characters. The maximum number of statements in a single compilation should not exceed 16383. This figure is the highest line number acceptable to the compiler and to the interactive debugging utilities. Programs more than 16383 lines long can of course be compiled, and listing segments created for them, but the lines above 16383 will not be accessible to the debuggers. This constraint is system-wide and independent of FORTRAN. In addition, the debuggers cannot display source lines located more than 262,144 characters from the start of the source segment.

Arrays and Common Blocks

For programs compiled with neither the la nor vla options, the limits of storage are: the combined size of the stack frame (which contains automatic) of the compilation unit cannot exceed 62,000 words, and the size of the combined linkage section (which contains static) cannot exceed 128K. (Note that the aggregate size of static for binding is 16K.)

When the la option has been selected, individual array sizes of 255K are supported in both static and automatic, with the compiler collecting storage items and allocating storage areas external to both the stack and area linker. When the vla_parm option has been selected, individual array parameters are limited to 2**24 words of storage. When the vla option is in effect, any numeric array or numeric common block can be up to 2**24 words in length, and character arrays can be up to 255K words. (The vla option turns on the la option as well, while the vla_parm option does not.)

Binder

The size of a program bound into a single segment by the bind command (including storage for all variables, local or common, that are bound with the program) must not be in excess of 261,120 words.

When you bind a program--a main program and all its associated subprograms--in a single object segment with the bind command, all the static variables must fit into 16,384 words of storage. (For more information on the binder, see Section 1 of this manual and the description of the bind command in the Commands and Active Functions manual.)

Stack Segment

The maximum stack frame size allowed by the FORTRAN compiler is 62,000 words of storage. (See Section 1 of this manual for a description of the stack segment.) The automatic variables in a program must therefore fit into 64K words of storage. The maximum stack size is 261,120 words of storage, but when you exceed 60,000 words of storage, the storage condition is signalled. Each time the storage condition is signalled an error message is printed. If the user types "start," the stack is extended by an additional 48K words. When the maximum stack size is exceeded, the user's process terminates.

Normal Storage vs. Large Arrays and Very Large Arrays

Multics FORTRAN release 10.2 provides two extensive enhancements to the intrinsic language storage capacity of FORTRAN. Two storage forms, Large Arrays (LA) and Very Large Arrays (VLA), provide the programmer with language addressable arrays that individually may be up to 2**24 words in length, and you may have aggregate storage available to your program in the range of 2**29 words.

Previous limits in force were:

Stack Frame Length

The stack frame, which holds all automatic variables, and compiler temporary variables, was previously limited to 62,000 words per compilation unit.

Internal Static Length

Internal static is part of the combined linkage section of the compilation/bind unit. The total length of a combined linkage section is 128K words. Static (save variables exist within static) is a portion of this area, which also contains initialization and linkage information. (Note that the aggregate size of static for binding is 16K.)

Common block length

Two forms of common block length were in force. Named common could be up to 255K (261120) words in length, while blank common was 255K-50 words in length (261070 words). Blank common was always allocated by the system at this maximum length.

The size limits for common, automatic, and static storage were for the aggregate as expressed by the total size for automatic of all temporaries and all automatic variables for the compilation. Static was the aggregate of all save variables, the linkage section, and the definition/initialization section for the compilation unit. Common was the aggregate storage size as assigned by the programmer for each named common block, or the total size of blank common within the compilation unit.

LARGE ARRAYS AND VERY LARGE ARRAYS

Large Arrays are a compiler collection of the variables for automatic and static storage into units that are individually within the hardware addressing limits of the processor (255K), but that may in total require an aggregate space larger than the previous automatic and/or static limits.

With the la option in force, all arrays become members of externally allocated storage areas and are managed by the compiler for storage collection and the runtime system for allocation, release, and initialization. There will be one or more segments allocated by a fortran_storage_manager_, either within the process directory or a specified "quota" directory, for each static or automatic variable. In the case of automatic variables, this will occur on the same basis as the allocation of stack frames--the automatic Large Arrays will be synonymous with the stack frame that "owns" them.

With the vla option in force, arrays may be declared to be larger than a single segment of storage, and thus incapable of being directly addressed by the hardware of the processor in a simple manner. This option enables the generation of extended addressing code to permit normal FORTRAN array addressing in a language-transparent fashion. The current size limit for each individual VLA is 2**24 words.

VLAs may exist for automatic, static, and common, with certain restrictions. A VLA cannot contain characters, since the EIS instruction set is incapable of crossing segment boundaries, and Multics IO system limits may restrict the size of a single binary IO operation to less than 255K words. The last restriction limits the ability to use implied do-loops on VLAs and prohibits their unsubscripted use in a binary IO list.

Automatic storage space allocated in LAs and VLAs is allocated upon entry to a compilation unit and is released upon exit from the compilation unit, either through a normal program return, a stop statement, or a release back to a command level below the start of the stack frame that "owns" the storage.

Static storage space allocated in LAs and VLAs is allocated upon first entry to a compilation unit and is released by the termination of the "owning" compilation unit or bind unit, through either a delete or terminate command, or the supported system subroutines.

Common storage space allocated in VLAs is allocated upon first entering a compilation unit and is released either by process termination, or by the delete_external_variables or set_fortran_common commands. At this time, there is no binder/linker support to enable binding and dynamic common creation of VLA common.

Permenent common storage cannot be used with VLAs because permanent common storage can only be one segment in length.

ACCURACY OF REAL NUMBERS

Multics' hardware operations may sometimes cause equivalent arithmetic expressions to produce slightly different computational results. (Two arithmetic expressions are mathematically equivalent if their mathematical values are equal for all possible values of their primaries.)

This can be demonstrated with double-precision numbers. The floating-point register (EAQ) represents a mantissa with 72 bits, whereas double-precision numbers in storage have only 63 bits of mantissa. Thus, the "same" number may have slightly different values in the register and in storage, though this difference will always be less than one part in 2**27 for single-precision numbers and less than one part in 2**63 for double-precision numbers.

OVERFLOWS IN INTEGER MULTIPLICATIONS

The hardware instruction that performs integer multiplication returns a double-precision integer result, but only the least significant half of that result is stored. Thus, in a simple assignment statement such as "i=j*k","i" will receive the expected value only if the product of "j" and "k" fits in a single-precision integer. Otherwise, "i" will get an unexpected value, but no error will be diagnosed.

If you are willing to accept a slight increase in the object size and execution time of your program, the compiler will insert extra code after each integer multiplication to see if the result exceeds single precision. When this option is chosen, the "fixedoverflow" condition will be signalled whenever the result of an integer multiplication is too large. (This is the same thing that happens when the sum of two integers exceeds single precision.) You may choose to ignore the condition and restart the program, in which case the most significant half of the product is merely discarded.

Checking for single-precision overflows in integer multiplication can be enabled by the -check_multiply control argument of the fortran command or by the check_multiply option of a %global or %options statement. This checking can be disabled by the -no_check_multiply control argument or by the no_check_multiply option. By default, checking is enabled unless optimization is requested. (See Section 3 above for a description of the fortran command and Section 1 of Multics FORTRAN for a description of the %global and %options statements.

SECTION 5

INPUT/OUTPUT IN MULTICS FORTRAN

INTRODUCTORY COMMENTS

It is useful, for most purposes, to look at input/output processing in Multics FORTRAN entirely in FORTRAN terms. The Multics FORTRAN implementation takes advantage of the design of the I/O system, which is intended to be, to a high degree, transparent--invisibly at work behind the scenes, doing input/output for you automatically. Your FORTRAN program doesn't know anything about the I/O system, and in many common programming situations you don't have to either. In fact, the Multics I/O system can be seen as a group of commands issued outside FORTRAN programs. These commands can be called in conjunction with FORTRAN programs (see "Connection Outside the Program," below for a discussion of that subject). More typically, input/output processing is handled with little or no direct intervention from you. The actual work of input/output, which can vary a great deal in nature, is performed by a subsystem called the FORTRAN runtime I/O routines, which is completely transparent. What you need to know is that it is the job of the FORTRAN runtime I/O routines to do input/output as directed by your program.

If you do not know anything about the Multics I/O system, and do not want to know anything about it, you can still do a lot with Multics FORTRAN. The next part of this section, in fact, is a guide to what you can do with little or no knowledge of the I/O system itself. Later parts cover the use of the open statement in FORTRAN, and hence describe some features of the I/O system; and finally there is a description of the command interface to the I/O system. If you are already familiar with the basics of Multics FORTRAN input/output, you may want to skip ahead to the subsection "Explicit Connection," which covers the use of the open statement and connection outside the program, i.e., from command level. If you are new to Multics, however, even a quick scan of the fundamentals will be instructive.

Fundamentals of Input/Output

The unit number in the FORTRAN program is the means of referring to a file from within the program. When FORTRAN was first developed, a unit was a particular physical device, such as a tape drive. Today, in Multics FORTRAN particularly, the unit has no ironclad association with a particular device. Even the units associated with devices by default can be explicitly connected to any files you choose. In general, then, a unit is something that has the property of being connected to or disconnected from a file. Connection is simply the association of the unit mentioned in your program with some physical storage medium.

Many systems require you to take explicit action in order to establish the connection between unit and file--often outside the program itself. While Multics offers various ways of connecting a unit outside the program, it is also possible for you to use the defaults in a way that requires neither external connection nor connection with open statements. If you are new to Multics or new to FORTRAN or both, you may find it useful to stick to the defaults and use only implicit connection at first. However, input/output tasks may require you to tell the system what to do in such detail that it is desirable to specify much of the connection explicitly; so you can also connect with an open statement or connect outside the program, as the standard requires. What follows describes the various ways to connect a unit in Multics FORTRAN, in terms of what you type at the terminal or what instructions you must include in your programs. There is only as much discussion of implementation details as is absolutely necessary for clarity. (If you want to know more about the implementation, especially of the FORTRAN runtime I/O routines, see the Multics FORTRAN manual, Section 10.) "Implicit Connection" below explains what happens when programs do input/output without using open statements to establish or change connection. "Explicit Connection" explains the use of the open statement, as well as preconnection from command level.

Implicit Connection

A unit is connected when a file is open and attached. In Multics, connection is either implicit or explicit. The FORTRAN Standard specifies that a unit is connected either by an open statement or by preconnection. Preconnection may be a result of job control language action or may be by processor-dependent defaults. Implicit connection is an example of the latter. Multics requires no job control action in connecting a unit. Instead, in implicit connection, Multics does the work for you--that is, the system takes all the steps necessary to connect a unit. (This method of connection is an extension of Standard FORTRAN.) Implicit connection takes place when there is no open statement in the program and no external, explicit preconnection.

(External preconnection is discussed below under "io_call attach for Device Independence.")

A program that contains data transfer statements, but no open statements, and which is not externally connected, results in implicit connection of the units specified by the unit numbers in the data transfer statements. The unit numbers 5 and 41 default to the terminal for input, and unit numbers 6 and 42 default to the terminal for output. Unit 0 is always considered preconnected to the terminal for input/output, and cannot be connected to any other device.

	input	output
unit number	5, 41	6, 42
default connection	tty	tty

The default connection in the absence of an open statement has the same effect as if the program contained an open statement of the proper form for the file or device which is associated with the unit number, and the type of I/O which is being requested in the I/O statement causing the default opening. The basic examples given later in the section show approximately what the "default open" looks like.

The really important point is that you need not take any action outside the program itself in order to connect the unit. If your program uses the default unit numbers, you need do nothing to connect the unit other than to include a data transfer statement in your program. But, to restrict yourself to the default unit numbers is little better than it would be to have no unit numbers at all. You have a hundred unit numbers available in all (0 through 99). You could create a catalogue in your hierarchy--for example a catalogue for input, and a catalogue for output, each with a hundred files 0 through 99. Multics provides greater flexibility still, since you can give files any names you choose, just so long as they conform to the naming conventions described in Section 1.

When you do not want to do input/output at the terminal--which is probably most of the time--it is not convenient to use default unit numbers. Implicit connection is possible, however, with all one hundred unit numbers. Implicit connection, when it is not to the terminal, is to a file in the storage system. Therefore when you use any unit number other than the five mentioned above, Multics connects by default to a

storage system file - a file in virtual storage, by which is usually meant a segment in the hierarchy.

The Use of Implicit Connection

There are two main reasons to connect a unit implicitly:

- You want to do input/output from the terminal directly and therefore use the default unit numbers

- Your program uses temporary files discarded at the end of a run

NOTE: If the reason your program does not contain an open statement is that it was written in accordance with the FORTRAN Standard of 1966, we don't suggest that you rewrite it merely to insert open statements. We do suggest that you avoid doing input/output via implicit connection, but rather connect the unit externally (as explained below under "io_call attach for Device Independence"). If you connect a unit outside the program you must disconnect it outside the program also.

Input Data Transfers

When implicit connection is established by a read statement, such as:

```
read (35, 100) i, j, k
```

if a file corresponding to unit 35 and containing data exists in your working directory, the read will occur automatically; but you must have done something to create the file prior to the read, and it must have a name that corresponds to the unit number--a name of the form file*nn* (with *nn* being the two-digit representation of the unit number). In the case illustrated in the read statement above, the file in the storage system would be file35. If no such file exists, the read fails, and Multics prints an error message that says you have attempted a read on a file that isn't to be found. If for some reason there is a file with the right name, but it is an empty file, a read statement referencing it will also be in error.

Output Data Transfers

The handling of output data transfers is symmetrical with that of input data transfers, but there is a difference. If a file corresponding to the unit does not already exist, Multics will create one, and data can be written into it. So files referenced by output data transfer statements need not exist

beforetimes. If no file exists to write the output to, the one Multics creates gets a name that corresponds to the unit number. If an appropriately named file exists before the unit is referenced in the program, the implicit connection takes place just as in the case of an input data transfer from an existing file, as explained above.

Explicit Connection

It may not always be convenient for you to write programs that depend on the defaults for implicit connection. Units used in existing programs written in accord with the ANSI Standard of 1966 (which did not allow open statements) may be connected from outside the program at command level. When you write the program using an open statement, you can specify the details of connection with great precision within the program itself.

There are, broadly speaking, two ways to establish explicit connection: 1) You can connect outside the program; 2) You can use the open statement within the program. Because connection outside the program requires greater knowledge of the Multics I/O system than other methods do, it is postponed until the end of this section. However,

IF YOUR PROGRAM MUST BE CONNECTED EXTERNALLY

because it was written that way, without open statements but requiring something other than the standard system input/output defaults, read "io_call attach for Device Independence" and "What's a I/O Switch?", below.

Using the Open Statement

The main use of the open statement is to connect a unit explicitly within the program. You are encouraged to use it for another reason: its presence in the source program can help to make your intentions clearer to any later programmers who need to maintain the code--such as yourself, two days from now.

Ordinarily, you will use the open statement to connect, explicitly, units that are not connected to the terminal by default. It is possible to use the open statement to connect any unit to the terminal, but in general:

USE IMPLICIT CONNECTION FOR TERMINAL I/O

and reserve explicit connection with the open statement for units that have no special default association.

Unit numbers other than 0, 5, 6, 41, and 42 are assumed to identify storage-system files.

The open statement provides a great deal of power and flexibility for input/output processing, but as a result it appears to be very complicated to use. In fact, in order to use all the powerful options of the open statement, you must have a more detailed knowledge of the Multics I/O system than it is within the scope of this manual to provide. Most FORTRAN programmers, however, do not need such detailed knowledge of the inner workings of Multics, and it is possible to get along perfectly well with a simple subset of open statement options, such as that described in the following pages.

*

WHAT IS IN THIS SUBSECTION

The examples and explanations that follow describe the forms of the open statement minimally necessary for getting the results obtained by the FORTRAN runtime I/O routines in doing implicit connection for each of the types of FORTRAN input/output. There are then examples of how to connect any FORTRAN unit (not only 0, 5, 6, 41, and 42) to the terminal, as well as how to connect units 5, 6, 41, and 42 to storage-system files. Next, there are some examples of how to use the open statement to connect tape files.

In most of these examples, files opened for write are opened with mode="inout", while files opened for read are opened with mode="in". (The meanings of the various mode values are described, along with other unit attributes, in the Multics FORTRAN manual, and will not be repeated here unless required in order to make an example completely clear.) This and other keywords specifying attributes are listed below, before the examples begin.

	WRITE	READ
DEFAULT MODE	inout	in

Shown above are the default and standard opening modes for implicit connection. In most of the examples below, except in opening units 5, 6, 41, and 42 for connection to the terminal, you can substitute "out" for "inout," or "inout" for "in," with no effect on the results. If you plan to do both reads and writes on the same file within one program, it is more efficient to open the file once for mode="inout" than to open it for one mode, close it, and then reopen it for the other. Moreover, if you want to open a file for inout, you can omit the mode="inout" specifier altogether, since "inout" is the default mode for open. (Some of the more rarely used I/O modules do not allow "inout" openings--an attempt to do it will get you an error message at runtime saying "invalid mode specified for device," and you will have to change the open statement to specify the particular mode required.)

In the examples below, the open statement includes specific unit numbers. In all cases, any other unit number might just as well have been used, and in your own programs you may use any valid number, except as noted and with the general exception that no other unit may be substituted for examples using unit number 0; nor may unit 0 be substituted in examples using other numbers. Unit 0 must be used to refer to the terminal.

Many examples show how to use the file specifier to designate a storage-system file by pathname when the default file name is not wanted. Use of a pathname to designate a file is usually not a good idea, for you will have to edit your program if the file is moved (for absolute pathnames) or if you change working directories (for relative pathnames). However, when the file is to be created, closed, and reopened later in the same run, and not used otherwise, it may be appropriate to designate a file by pathname. Another use of a pathname to designate a file is in applications in which it is certain that the file will stay in the same place forever. Finally, since the value given with the file specifier can be a character expression, it can be used in cases where the FORTRAN program asks for the name of a file, reads it into a character variable, and then uses that variable as the file name (file=char_variable_containing_name).

In examples that show how to turn on the standard carriage-control conventions, the full form is used. If the + (overprint) carriage control is not used in the program, the defer specifier can be omitted.

The err and iostat specifiers can be used in any open statement except when specifying unit number 0, and these specifiers do not appear in any of the examples.

If the associated file exists and is not empty, whether the program reads from it or writes into it, the attributes expressed in the open statement must correspond to the actual attributes of the file. The main thing to keep in mind as far as your FORTRAN program is concerned is that the file must be defined with attributes that are consistent with the type of input/output processing you plan to do. You would not, for example, attempt to do unformatted input/output on the terminal, and an attempt to do so would produce an error message from the FORTRAN runtime I/O routines. Consistency of attributes does not in all cases mean identity--a full knowledge of the vfile_ I/O module (beyond the scope of this manual) would provide you with ways to change the apparent attributes of a file for differing purposes. It should also be mentioned here that references, below, to the vfile_ I/O module and its control orders describe operations performed by these runtime I/O routines. The information is offered for clarity. Nothing in the sections on the open statement requires you to take explicit action with respect to vfile_ or any other I/O module. The attach specifier is supplied with the open statement, for use by those with sufficient knowledge of the I/O system to use it, but no information about its use is provided in this manual.

It is recommended that in general you use the open statement when writing new programs, since it makes explicit in the source what you intend your program to do. It will in many instances do little or nothing else; but while the presence of the open statement may not materially alter the behavior of a program in execution, the open statement is helpful to you for future reference and will be similarly helpful to the next generation of programmers to use your programs. The open statement supplements the implicit method of connection in essentially two ways: 1) it allows you to include in your program explicit instructions to do what the FORTRAN runtime I/O routines would otherwise do for you; 2) it allows you to alter the default connection.

A complete list of specifiers available with the open statement appears, with explanations, under "I/O Control Statements" in Section 5 of Multics FORTRAN. The specifiers themselves are listed for quick reference below.

access	form
attach	iostat
binary stream	ioswitch
blank	mode
carriage	prompt
defer	recl
err	status
file	unit

The following are some examples of standard forms of open statements, showing which forms to use for what kind of connection.

TERMINAL READ/WRITE (UNIT 0)

Unit 0, which is used in print and input statements, and in read statements that do not specify a unit number, is the easiest to use. To use the open statement to open unit 0 explicitly in the same way the FORTRAN runtime I/O routines do it in implicit connection, use the statement:

This page intentionally left blank.

```
open (0)
```

The defer, carriage, and prompt specifiers are the only others that may appear. To open unit 0 with standard carriage control conventions, use:

```
open (0, defer=.true., carriage=.true.)
```

TERMINAL READ (UNITS 5 AND 41)

When you use one of these units in a read statement, the FORTRAN runtime I/O routines assume that the unit number refers to the terminal unless you open the unit explicitly for connection to some other file or device. To use the open statement to open unit 5 explicitly in the same way that the FORTRAN runtime I/O routines do it in implicit connection, use an open statement of the form:

```
open (5, mode="in", form="formatted", access="sequential")
```

Substitute 41 for 5 to open unit 41.

You may use the prompt specifier (prompt=.true.) if prompting is required.

TERMINAL WRITE (UNITS 6 AND 42)

When you use one of these units in a write statement, the FORTRAN runtime I/O routines assume that it refers to the terminal unless you explicitly use the open statement to connect it to some other file or device. To use the open statement to connect unit 6 explicitly the same way that the FORTRAN runtime I/O routines do it in implicit connection, use an open statement of the form:

```
open (6, mode="out", form="formatted", access="sequential")
```

The defaults for these units are defer=.false. and carriage=.true., which means that all the standard carriage control characters except + will work. For full carriage control, including +, use an open statement of the form:

```
open (6, mode="out", form="formatted",
      access="sequential", defer=.true.)
```

To completely disable carriage control, use an open statement of the form:

```
open (6, mode="out", form="formatted",
      access="sequential", carriage=.false.)
```

Substitute 42 for 6 to open unit 42.

FORMATTED SEQUENTIAL I/O TO STORAGE SYSTEM FILES

Formatted sequential input/output to a storage system file is what the FORTRAN runtime I/O routines assume you want when you use a simple formatted read or write such as:

```
read (45, 100) i, j, k
```

NOTE: The examples below apply to all units except 0, 5, 6, 41, and 42.

For input, use an open statement of the form:

```
open (45, mode="in", form="formatted")
```

For output, use an open statement of the form:

```
open (45, mode="inout", form="formatted")
```

You may include the specifier access="sequential" for clarity if you wish. Carriage control is normally disabled on storage system files, but you can turn it on by using the specifiers defer=.true. and carriage=.true. Carriage control is not normally useful or desirable when writing to a storage system file, however.

The examples given above refer to a file in your working directory named file45. If you open for out or inout when the file does not exist, a vfile_ stream unstructured file is created with that name. The vfile_ control order -extend is used to allow you to append new output to the file by positioning first to the end (that is, by reading until you get to the end of the file) and then writing; you should open the file for mode="inout" if you intend to make use of this feature.

If you wish to open a file under a name other than file45, you may use the specifier file="<pathname>", where the pathname is the relative or absolute pathname of the desired file, as in

```
open (45, mode="in", form="formatted",
      file=">udd>Proj>Person>datafile")
```

You may use the recl specifier to define a maximum record length if you so desire. If you use the recl specifier when the file exists, the file must already have a defined maximum record length, and the length specified in the open statement must match the length already associated with the file. The file is opened with the vfile_ controls -no_end and -extend, so that information already in the file will not be destroyed by the opening. You

must position to the end of file as described above if you want to append data to the file.

If you use the recl specifier when the file does not exist, and it is being opened for out or inout, it is created as a vfile_ blocked file with a maximum record length as specified. (See the MPM Subroutines manual for a complete description of the vfile_ I/O module. If you do not understand the relationship between files and I/O modules, see "What's a I/O Switch?" below.)

UNFORMATTED SEQUENTIAL I/O TO STORAGE SYSTEM FILES

Unformatted sequential I/O to a storage system file is what the FORTRAN runtime I/O routines assume when you use a simple unformatted read or write, such as:

```
read (47) i, j, k:
```

This form of read statement gives the most compact representation of data for temporary files, but it is machine-dependent and recommended only if the data need not be transported from Multics to other systems. This form of the read statement also places a heavier burden on the programmer, since there is no checking of data types (see the Multics FORTRAN manual for fuller details). The examples below apply to all units except 0, 5, 6, 41, and 42.

For input, use an open statement of the form:

```
open (47, mode="in")
```

For output, use an open statement of the form:

```
open (47, mode="inout")
```

You may use the specifiers access="sequential" and form="unformatted" for the sake of clarity. You may not use the carriage, defer, or prompt specifiers.

The example given refers to a file named file47 in your working directory. If you open a file for out or inout when it does not exist, a vfile_ sequential file is created with that name. When the file already exists, the vfile_ control order -extend is used, so that you can add output to the file by positioning first to the end (that is, by reading until you get to the end of the file), and then writing; you should open the file for inout if you plan to do this.

If you wish to open a file with a name other than file47, you may include the specifier file="<pathname>", where pathname is the absolute or relative pathname of the desired file, as in

```
open (47, mode="inout", file="mydata")
```

which refers to a file named mydata in your working directory.

You may use the recl specifier to specify a maximum record length. If you use the recl specifier when the file exists, the file must already have a defined maximum record length, and the length specified in the open statement must match the length already associated with the file. The file is opened with the vfile_ controls -no_end and -extend, so that information already in the file will not be destroyed by the opening. You must position to the end of file as described above if you want to append data to the file.

If you use the recl specifier when the file does not exist, and you are opening it for out or inout, it is created as a vfile_ blocked file with a maximum record length as specified. (See the MPM Subroutines manual for a complete description of the vfile_ I/O module. If you do not understand the relationship between files and I/O modules, see "What's a I/O Switch?" below.)

DIRECT ACCESS FORMATTED I/O TO STORAGE SYSTEM FILES

Direct access formatted I/O is what the FORTRAN runtime I/O routines assume when you use direct access formatted reads or writes such as:

```
read (49'2, 100) i, j, k
```

NOTE: The examples below apply to all units except 0, 5, 6, 41, and 42.

For input, use an open statement of the form:

```
open (49, mode="in", access="direct", form="formatted")
```

For output, use an open statement of the form:

```
open (49, mode="inout", access="direct", form="formatted")
```

The example refers to a file named file49 in your working directory. If you open the file for out or inout when it does not exist, a vfile_ keyed_sequential indexed file is created with that name. If the file exists, the vfile_ control -extend is used so that the data in it will not be destroyed by the opening.

If you wish to open a file with a name other than file49, you may use the specifier file="<pathname>" where pathname is the relative or absolute pathname of the desired file, as in:

```
open (49, mode="in", access="direct",
     form="formatted", file=">udd>Proj>Person>data")
```

which refers to a file named data in the directory >udd>proj>Person.

You may use the recl specifier to specify a maximum record length, if you desire. If you use the recl specifier when the file exists, the file must already have a defined maximum record length attribute, and the length specified in the open statement must match the length already associated with the file. The file is opened with the vfile_ controls -no_end and -extend so that information already in the file will not be destroyed by the opening.

If you use the recl specifier when the file does not exist, and it is being opened for out or inout, it is created as a vfile_ blocked file with a maximum record length as specified. (See the MPM Subroutines manual for a complete description of the vfile_ I/O module. If you do not understand the relationship between files and I/O modules, see "What's a I/O Switch?" below.)

The use of recl is recommended if all records are known to be about the same size, since blocked files can be manipulated more efficiently than indexed files. If the records differ in size, use of recl will waste space but speed processing. You will have to decide for yourself which alternative gives better overall use of resources for each particular case. If the maximum record length is unknown, recl cannot be used.

DIRECT ACCESS UNFORMATTED I/O TO STORAGE SYSTEM FILES

Direct access unformatted I/O is what the FORTRAN runtime I/O routines assume when you use a direct access unformatted read or write, such as:

```
read (51'4) i, j, k
```

This version of the read statement gives a fairly compact representation of data for temporary files, but it is machine-dependent and so is recommended only if the data will not be transported from Multics to other systems. This read statement also places a heavier burden on the programmer, since there is no checking of data types (see the Multics FORTRAN manual for fuller details).

NOTE: The examples below apply to all units except 0, 5, 6, 41, and 42.

For input, use an open statement of the form:

```
open (51, mode="in", access="direct")
```

For output, use an open statement of the form:

```
open (51, mode="inout", access="direct")
```

You may add the specifier form="unformatted" for clarity. You may not use the carriage, defer, and prompt specifiers.

The example above refers to a file named file51 in your working directory. If you open a file for out or inout when it does not exist, a vfile_ keyed_sequential indexed file is created with that name. The vfile_ control order -extend is used so that if the file already exists the data in it will not be destroyed by the opening.

If you wish to open a file with a name other than file51, you may use the specifier file="<pathnname>" where pathname is the relative or absolute pathname of the desired file, as in

```
open (51, mode="in", access="direct", file="my_data")
```

which refers to a file named my_data in your working directory.

You may use the recl specifier to specify a maximum record length, if you desire. If you use the recl specifier when the file exists, the file must already have a defined maximum record length attribute, and the length specified in the open statement must match the length already associated with the file. The file is opened with the vfile_ controls -no_end and -extend so that information already in the file will not be destroyed by the opening.

If you use the recl specifier when the file does not exist and is being opened for out or inout, it is created as a vfile_ blocked file with a maximum record length as specified. (See the MPM Subroutines manual for a complete description of the vfile_ I/O module. If you do not understand the relationship between files and I/O modules, see "What's a I/O Switch?" below.)

The use of recl is recommended if all records are known to be about the same size, since blocked files are processed with

less cpu time than indexed files. If the records differ in size use of recl will waste space, but speed processing. You will have to decide for yourself which alternative gives better overall use of resources for each particular case. If the maximum record length is unknown, recl cannot be used.

BINARY STREAM FILES

Binary stream files are a Multics-specific file type. They provide a way of getting 36-bit words of data from a file in the storage system to your program and back. The use of binary stream files is not recommended--they are, moreover, usually unnecessary, in that there is little you can do with them that you can't do with FORTRAN unformatted files. In any case, only files created as binary stream files by FORTRAN should be read as binary stream files. Essentially, if you have to look up the information on how to open a binary stream file, you shouldn't be using it. However, for those of you who insist upon finding new and different ways to get yourselves in trouble, here goes:

You may open a binary stream file for either direct or sequential I/O. If you open it for direct I/O, each word is considered a separate record.

For sequential input, use an open statement of the form:

```
open (99, mode="in", binary stream=.true.)
```

For direct input, use an open statement of the form:

```
open (99, mode="in", access="direct", binary stream=.true.)
```

For sequential output, use an open statement of the form:

```
open (99, mode="inout", binary_stream=.true.)
```

For direct output, use an open statement of the form:

```
open (99, mode="inout", access="direct",
      binary stream = .true.)
```

The examples refer to a file named file99 in your working directory. If you open a file for out or inout when it does not exist, a vfile_ unstructured stream file is created with that name. If the file already exists, the vfile_ control order -no_trunc is used, so that information in the file will not be destroyed by the open statement. This allows you to append new output to the file, by positioning to the end (that is, by reading until it gets to the end of the file) and then writing;

the file should be opened for inout if you intend to use this method.

If you wish to open a file with a name other than file99, you may use the specifier file="<pathname>", where pathname is the relative or absolute pathname of the desired file, as in:

```
open (99, mode="in", binary stream=.true., file="bad_idea")
```

which refers to a file named bad_idea in your working directory.

CONNECTING NONSTANDARD UNITS TO THE TERMINAL

The easiest way to connect a nonstandard unit (that is, a unit other than 0, 5, 6, 41, or 42) to the terminal is to use

```
open (55, form="formatted", io_switch="user_i/o")
```

This opens FORTRAN unit 55 connected to the terminal for input and output. The specifiers access="sequential" and mode="inout" may be added. If you want to open the unit for input only, you may specify mode="in", in which case io_switch="user_input" may be used; if you want to open only for output, you may specify mode="out", in which case the io_switch may be either "user_output" or "error_output". All four named io_switches usually refer to the terminal, but if you are using file_output or discard_output commands the distinction may matter.

If you want to be truly fancy, you can use

```
open (55, form="formatted", attach="syn_ user_i/o")
```

Again, you may use the other io_switch names if mode is set to correspond.

When you open a nonstandard file as connected to the terminal, carriage control for terminal I/O is disabled. To get standard carriage control, add the specifiers carriage=.true. and defer=.true. to the open statement. You may also use the prompt attribute if you desire.

CONNECTING A DEFAULT TERMINAL UNIT TO A FILE

The default terminal units are 0, 5, 6, 41, and 42. Unit 0 must always be connected to the terminal; units 5, 6, 41, and 42 may be connected to files. Heaven only knows why you would want to connect one of these to a file, as there are 95 other units available. On the other hand, since there's always someone, this section describes how to do it. Simply use the normal form

described above for the particular type of I/O you want, using the unit number (5, 6, 41, 42) desired, but add one of the specifiers file="<filename>" or attach="vfile_ <filename>", where filename is the name of the file you want. This will let FORTRAN know that you want the unit to refer to a file rather than to the terminal. Note that the normal rules for carriage control for these unit numbers apply. If you use unit 6 or 42, and you don't want carriage control, or the opening is not for sequential formatted output, include the specifiers defer=.false. and carriage=.false. If you are opening unit 6 or 42 for sequential formatted output and you want carriage control, include the specifiers defer=.true. and carriage=.true. For example:

```
     open (5, access="direct", form="unformatted", mode="in",
          attach="vfile_ file05")

     open (6, access="sequential", mode="inout",
form="formatted",
           carriage=.false., defer=.false.,
           file=">udd>myproj>me>foo")
```

CONNECTING 6 OR 42 FOR TERMINAL INPUT, 5 OR 41 FOR TERMINAL OUTPUT

Don't be ridiculous. If you really want to do this, you'll have to figure it out for yourself. We won't be a party to such foolishness.

CONNECTIONS TO TAPE FILES

Each of the tape I/O modules is different, and has different capabilities. The examples given below are intended as skeletons only, not as complete cookbook statements to suit every need. Before using a tape I/O module you should read the documentation for it in the MPM Peripheral I/O Guide, to determine whether any additional arguments are needed in the attach description. These are to be placed in the examples where the marker <ADDED_ARGS> appears.

Note that some of the tape I/O modules impose restrictions upon you beyond those imposed on a given opening by FORTRAN. For example, tapes can be opened only for sequential I/O. As this is FORTRAN's default, the access specifier can be omitted. In addition, at the time of this writing, tape_nstd_ requires that all records be an integral number of words (a multiple of 4 characters) long; tape_ibm_ and tape_ansi_ do not allow certain positioning operations, in particular, the backspace operation; and tape_ibm_, tape_ansi_, and tape_mult_ do not allow inout opening. This list of restrictions is not meant to be exhaustive, and may change in any case, so you should check MPM Peripheral I/O.

The form specifier may be given as either "formatted" or "unformatted," with unformatted the default. Formatted tape_ansi_ is recommended if you intend to transport the tape to a non-Multics system.

For input use an open statement of the form:

```
open (10, mode="in", attach="tape_ansi_ vol001 -nm file_name
      <ADDED_ARGS>")

open (10, mode="in", attach="tape_ibm_ vol001 -nm file_name
      <ADDED_ARGS>")

open (10, mode="in", attach="tape_mult_ vol001
<ADDED_ARGS>")

open (10, mode="in", attach="tape_nstd_ vol001
<ADDED_ARGS>")
```

For output, use an open statement of the form

```
open (11, mode="out", attach="tape_ansi_ vol001 -nm
file_name
      -cr -ring <ADDED_ARGS>")

open (11, mode="out", attach="tape_ibm_ vol001 -nm file_name
      -cr -ring <ADDED_ARGS>")

open (11, mode="out", attach="tape_mult_ vol001 -write
      <ADDED_ARGS>")

open (11, mode="inout", attach="tape_nstd_ vol001 -write
      <ADDED_ARGS>")
```

Using the Inquire Statement

Some files cannot be read or written by direct access in a FORTRAN program. The inquire statement enables you to determine, from within the program, the opening modes for a file. When a file is already connected to a unit, the inquire statement can ascertain the attributes of the unit. Use of the inquire statement is explained in detail in the Multics FORTRAN manual.

io call attach for Device Independence

Full coverage of this subject is beyond the scope of this manual. A quick overview should be given, though. For this example, formatted sequential input is assumed. The methods are the same for other types of I/O, but you must make sure that the file or device you attach externally can support the type of I/O you intend to do. For example, you cannot open a unit for direct_access I/O if it has been attached to a tape file and you cannot open a unit for unformatted I/O if it has been attached to the terminal.

In the examples for tapes the flag <ADDED_ARGS> means that you may need to supply additional arguments. You should read the tape module documentation in MPM Peripheral I/O Manual before using a tape module; that will tell you what the possible arguments are.

This page intentionally left blank.

One feature of the Multics FORTRAN runtime I/O package is that when it closes a file it only undoes those things which it did. Thus, if you externally attach unit 20 to a vfile_ named >udd>proj>me>foo, all FORTRAN programs which you run which use unit 20 will reference that same file, until you detach unit 20 using the io_call command, until your run unit (if you're using them) ends, or until your process terminates. Be sure you detach anything which you attach, unless this file-sharing is what you want.

All the examples use unit 20. You may substitute any other unit you like. The file and attach specifiers cannot be used in the open statement inside the program if you attach the file externally.

Example 1: Using FORTRAN unit names

If the program contains the statement:

```
open (20, form="formatted", mode="in", prompt=.true.)
```

or uses unit 20 for formatted sequential input with implicit opening, then executing the command:

```
io_call attach file20 syn_ user_i/o
```

before running the program will cause read statements referencing unit 20 to take input from the terminal (with prompting, if the open statement was used); the command:

```
io_call attach file20 vfile_ >udd>proj>me>data
```

will cause read statements referencing unit 20 to take input from the storage system file named >udd>proj>me>data; and the command:

```
io_call attach file20 tape_ansi_ vol475 -nm data
        <ADDED_ARGS>
```

will cause read statements referencing unit 20 to take input from the file named data on the ANSI Standard format tape with the label vol475. Don't forget to say:

```
io_call detach file20
```

when you're done.

Example 2: Using mnemonic io switch names

If you don't want to have to remember what file20 is, as compared with file19 and file21, you can use more mnemonic names,

by using the ioswitch specifier in the open statement. Assuming the same files as example 1, you could say, in the program:

```
open (20, form="formatted", mode="in", prompt=.true.,
      ioswitch="input_data")
```

Then, to take data from the terminal, you would use the command:

```
io_call attach input_data syn_ user_i/o
```

before running the program. To use the storage system file, use:

```
io_call attach input_data vfile_ >udd>proj>me>data
```

Finally, for the tape file, use:

```
io_call attach input_data tape_ansi_ vol475 -nm data
        <ADDED_ARGS>
```

When you do it this way, references to unit 20 in other FORTRAN programs won't get the same connection (unless the other programs also open with ioswitch="input_data"); but you should still remember to say:

```
io_call detach input_data
```

when your program ends.

A vfile_ or tape file need not exist at the time you execute io_call to attach to it, so this method can also be used to gain device independence for files which you plan to create in the run. If you're creating the file on a tape, remember to check MPM Peripheral I/O to see what arguments you need. In particular, you need at least -ring (for tape_ansi and tape_ibm_) or -write (for tape_mult_ and tape_nstd_) before you can write on the tape at all. vfile_ also can take added arguments, but in most circumstances you'll meet in FORTRAN you shouldn't need them. If you are curious, see the vfile_ documentation in MPM Subroutines.

And <u>don't</u> forget "io_call detach".

io_call open for Complete External Connection

It is possible to use the io_call command to open a file outside the program. You may wish to perform all the steps of connection from command level, using the io_call attach and io_call open commands. (Use io_call open only after first

invoking io_call attach; you cannot open a file that has not been attached, any more than you can open a door without a handle. And watch your parking meters!) If you intend to do the I/O by hand, however, it is imperative that you be fully familiar with the details of the FORTRAN runtime I/O routines, described in Section 10 of the Multics FORTRAN manual, as well as with the Multics I/O system as described in the MPM Reference, before using the io_call open command.

Although a complete discussion of the io_call open command is beyond the scope of this manual, an introductory sketch is called for. What is offered below presents the bare essentials only.

You might invoke the io_call open command for input/output, as follows:

```
io_call open user_io sequential_input_output
```

This invocation of io_call open is the one appropriate for formatted sequential read or write statements in your program. An open statement in the same program may specify the mode attribute as <u>inout</u>, <u>in</u>, or <u>out</u>.

Why do you have the option of giving two distinct and even slightly differing mode specifications for one program, one in the io_call open command and one in the open statement within the program itself? The question must receive two answers.

First, for most programming purposes, it is completely unnecessary to use the io_call open command in conjunction with the open statement, since what the open statement specifies, will be faithfully carried out by the FORTRAN runtime I/O routines. So why bother? In explicitly specifying both attachment and opening through the medium of the io_call command, you get direct control of and the steps of connection, but you also greatly increase your chances of making a programming error. The general philosophy behind these comments is "What Multics can do for you, let it do for you." That's what Multics is all about, so the first answer to the question of why you have all these options is, "never mind, don't do it." The second answer is more pragmatic. If you must open with the io_call open command, use an opening mode that is identical to or broader than what your program specifies in a data transfer or an open statement. If your program does input and output on the same file, open the file for input/output. That is the answer to why the modes, as specified in the io_call open command and in the open statement, may differ.

To get an idea of the difficulty associated with opening the file with the io_call open command, consider the following example. You invoke io_call open more narrowly than in the first example above, i.e.:

```
io_call open user_output sequential_output
```

or

```
io_call open user_input sequential_input
```

In either case the open mode is too restrictive to permit an opening within your FORTRAN program for ANY of the other modes. The first case is appropriate for a program that does output only; the second for a program that does input only. You could use two invocations of io_call open, one for input, one for output, on two different files. The mode specified for a file outside the program, however, must always either be less restrictive than or the same as the modes specified for a file inside the program.

MAKE YOUR OPENING MODES CONSISTENT

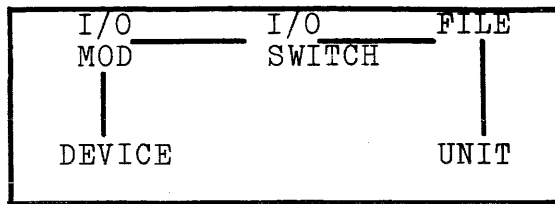
The opening mode specified in the io_call open command must not be more restrictive than the mode specified in the corresponding open statement, and it must not be in direct conflict with it either (out vs. in, for example.)

Also, if the open statement specifies a more restrictive mode than that specified in the io_call open command (in vs. inout, for example), subsequent data transfers on the file are limited to the mode specified in the open statement. The fact that the mode specified in the io_call open command is less restrictive than that specified in the open statement, that is to say, does not determine the mode for your program. To change the mode specified in an open statement you must include a close statement, and then a new open statement specifying the new mode.

Finally, it should be clear that you are not encouraged to use the io_call open command. But if you must use it, remember to use io_call close at the end!

WHAT'S A I/O SWITCH?

The Multics I/O system, rather than connecting devices directly to files, connects devices to files through the medium of I/O switches and I/O modules.



Each of these items has a name, and connection may be regarded as the association--in your programming environment, and yours only--of one such collection of names. The unit number in the FORTRAN program is associated with the name of a file or a device, which is in turn associated with the name of an I/O switch. This association is called attachment. The I/O switch (and hence the associated file or device) is associated by name with the I/O module, and the I/O module with some device. (From the viewpoint of the Multics I/O system, the storage system--or more precisely, any segment in virtual storage--is a device. Devices are physical storage of some kind.)

In simplest terms, the unit number is the number used in a FORTRAN I/O statement to specify a device or a file. The device, or file is a real device such as your terminal, or a file in the storage system or on some other medium such as tape. The I/O module is a system program designed to operate on data in or on such a file or device, and to provide a standard interface to the user I/O routines--in this case, the FORTRAN runtime support package. Aside from knowing that these I/O modules exist, you need not normally know anything more about them. The io_switch can be viewed as nothing more than a sort of note from Multics to itself, telling it which unit number in your program refers to which I/O module/device or I/O module/file combination, and keeping track of the state of the file or device and of some of its characteristics.

SECTION 6

CONVERSION TO FORTRAN 77

FORTRAN 77 ON MULTICS

The FORTRAN 77 language is being implemented on Multics by extending the current Multics FORTRAN compiler. The ultimate goal of this extension is to produce a compiler that accepts programs that conform to the FORTRAN 77 standard, that continues to accept existing Multics FORTRAN programs with no modification, and that attempts to smooth the transition from 1966 FORTRAN to FORTRAN 77.

To help meet this goal, two new options have been introduced--ansi66 and ansi77. Any particular program unit is compiled with one of these options in effect. Program units compiled under the ansi66 option will be interpreted as they have been interpreted by Multics FORTRAN in the past. Program units compiled under the ansi77 option will be interpreted according to the FORTRAN 77 standard wherever it differs from the existing Multics FORTRAN language. The default is ansi66.

It is important to note that the distinction between ansi66 and ansi77 does not affect all new features of FORTRAN 77, nor does it affect all features of Multics FORTRAN that are extensions to the 1966 standard. Instead, the distinction serves only to control the interpretation of constructs that have different, incompatible meanings or implementations in Multics FORTRAN and FORTRAN 77.

As many features from Multics FORTRAN and from FORTRAN 77 as possible are available under both the ansi66 and ansi77 options. The benefit of this approach is twofold. First, it allows existing programs to use some of the new features in FORTRAN 77 without full conversion. Second, it allows programs to be easily converted to FORTRAN 77; only the particular constructs that are incompatible need be changed.

CONVERSIONS

Certain FORTRAN 77 features differ from Multics FORTRAN in such ways that existing programs must be converted in order to run under the ansi77 option with the same semantics. Described below are six of the most useful conversions to make. All the differences between the ansi66 and ansi77 options are listed in Appendix B of the Multics FORTRAN manual.

Character-Mode Variables in Common Blocks

Under the ansi77 option, character-mode variables cannot be mixed with variables of other modes in a common block. Common blocks in ansi66 programs that contain both character and noncharacter data must be split into two separate common blocks, one for character data and one for noncharacter data.

Equivalencing Character-Mode Data

Under the ansi77 options, character-mode variables cannot be equivalenced with variables of other modes. Such equivalencing in ansi66 programs should be replaced with explicit assignment statements. For complex and double-precision data, the corresponding character variable should be given the length 8. For integer, real, and logical data, the corresponding character variable should be given the length 4.

Whenever the storage in question is to be viewed as noncharacter data, the program should explicitly assign the character value to an integer, real, double-precision, or complex variable. Whenever the storage is to be viewed as character data, it may be inspected directly.

Default Character-String Length

Under the ansi77 option, the default length for character variables has changed from 8 to 1. To avoid ambiguity, all character statements in the program should be inspected to ensure that every character mode variable has an explicitly declared length.

Packed Character-String Layout

The representation of character data in storage is different under the ansi77 option. In the ansi66 implementation, all character variables and array elements are stored as aligned character strings, that is, starting on a word boundary in the computer memory. In the ansi77 implementation, character variables may be stored as unaligned character strings; that is, each array element follows the preceding element with no intervening padding. Thus, elements may begin at character positions that are not word boundaries. This change will most seriously affect programs that use permanent common blocks (those whose names end with "$") or unformatted files that contain character data and that were written by an ansi66 program.

An ansi77 program can access character data in ansi66 format as follows:

1) For each character datum in ansi66 format, the ansi77 program should declare a corresponding character datum. The length of the character datum in the ansi66 program should be the smallest multiple of 4 that is greater than or equal to the length declared in the ansi66 program. For example, a character*15 variable becomes character*16, and a character*32 variable remains character*32.

2) The next step is to use the substring notation wherever an ansi77 variable is used to access ansi66 data. If the variable is declared character*15 in the ansi66 program, the ansi77 program should reference a character*16 variable with the substring notation (1:15).

This technique can be used in converting old format data to the new format by reading the data as described above and writing it to a new file or common block with an ansi77 program.

Zero-Trip Do Loops

For ansi66 programs being converted to ansi77, each do loop must be examined. If the logic of the program depends on the loop being executed at least once in all circumstances, the final loop value should be changed to use the max or min intrinsic function. That is because under the ansi77 option loop counts that are zero or negative cause the loop to be skipped entirely, whereas under ansi66, loops are always executed at least once.

For an example of this conversion,

```
do 100 I = J,K
```

might be changed to

```
do 100 I = J, max (J,K)
```

to ensure that the loop is executed once. If you know the increment to be negative, the min intrinsic function should be used instead of max.

Blank Lines

Under the ansi77 option, blank lines are treated as comment lines and thus ignored. The ansi66 option treats them as initial lines. Hence, when a blank line precedes a continuation line, the latter is treated as a continuation of the blank line

The following program demonstrates this difference:

```
        % options card;
        program blank_line
        integer foo, foogoto5, k
        data foo /66/, foogoto5 /77/
        k = foo

      & goto 5
100     format ("This was compiled with the ansi", i2, "option.")
  5     write (6,100) k
        step
        end
```

If you compile this program with the ansi66 option, the blank line is interpreted as an initial line, so the "&goto 5" is a continuation of the blank line and k is set to foo, i.e., 66. If you use the ansi77 option, the blank line is ignored, and the value of k is set to foogoto5, i.e., 77, because the "goto 5" is treated as a continuation of the line "k = foo".

To convert ansi66 programs to ansi77, take out the continuation marker (e.g., "&") so that the line so marked is treated as an initial line.

APPENDIX A

DEBUGGING

This section covers the debugging of FORTRAN programs in the Multics system, with the emphasis on interactive debugging with the probe command.

First, there is a description of the specifications for a program that plays tic-tac-toe, followed by discussion of the way this program works. Next follows a version of the source program, with comments. The first version of the program is incorrect and doesn't play fair. Following this version of the program is a script of a debugging session. The script provides an example of the use of the probe command and some of its major requests, but you are encouraged to read the description of the probe command in the MPM Commands as well. Finally there is a version of the corrected source program.

SPECIFICATIONS FOR TIC-TAC-TOE PROGRAM

The program plays optimally--that is, it wins if it is possible to win, otherwise it draws. (Either player can always force a draw in tic-tac-toe.)

After each machine move the board is printed, showing the moves on the board.

The cells of the board are numbered from 1 to 9, from the upper left hand cell to the lower right hand cell. The numbered board is printed at the first invocation of the program, and not again.

Your moves are carefully checked for validity. A valid move must be in a cell where there is not a move already, and it must be a digit from 1 to 9. End of file causes a draw.

The program detects wins or draws, prints the winner, and the final state of the board.

Only one game is played at each invocation of the program.

HOW THE PROGRAM WORKS

A listing of the source program appears below. The program consists of three program units, main_, mover, and won. There is also a block data subprogram.

The main program performs all the input/output for the program, checks the player's moves, and calls the subprograms, mover and won. Its first action, if

this is the first invocation of the program in the player's current environment, is to print the numbering scheme and the instructions. It then enters the main loop at statement 5. This loop reads the player's moves, validates them, checks for a win by the player, gets a machine move from the subroutine, checks for a win by the machine, displays the board, and continues in this vein until the game ends. Statements executed for a draw (97), a win by the player (99), a win by the machine (100) print the result (who wins, or "Cat's game" if there is a draw) and the final board.

A few details: the board is represented as an integer array (board), with one array element for every cell. The state of a cell (empty, x, or o) is recorded in the corresponding array element, which is assigned a different integer for each state. The particular integer values used to represent the states are chosen for convenience and have no significance apart from their use in selecting array elements elsewhere in the program.

The array is stored in the named common block tic_tac_toe, where all the program units can reference it. Note, also, the complicated format (statement 51) used to print the board.

The subroutine, mover, sets its parameter to the number of the square into which the machine moves, or to zero if there is no empty square left (the game is a draw). Two two-dimensional arrays, paths and paths_thru_cell, contain invariant information about the game.

The term path means three cells in the same row, column, or diagonal. The array, paths, defines the eight paths in the game by giving the cell numbers of the cells in the path, and the array, paths_thru_cells, gives the path numbers of the paths that pass through each cell. The center cell (5) has four paths through it, the corner cells have three, the rest two; zero is used to indicate the absence of a path.

The first group of statements in the subroutine, a loop ending at 10, calculate the pathsum of each path. The pathsum is an integer value that gives the state of each path. A unique weight is assigned to each state of a cell. The pathsum of a path is the sum of the weights of each of its cells. The values used to represent the weight of each state of a cell can be chosen arbitrarily, as long as each interesting state of a path has its own unique value. The paths of interest are those that have: two o's and no x, one o and no x, one x and no o, and two x's and no o.

The subroutine determines what move to make by trying five strategic rules in order of priority (they appear in order below). If a rule is applicable, that determines the move the machine makes. If not, the program goes to the next rule. Except in the case of a draw, at least one of the first four rules will be applicable. The rules are:

1. Search for a path with two o's and no x. This path can be completed to give the machine a win. A three-statement loop (98-100) must be executed to find the empty cell, since pathsum does not tell which cell of a path is empty.

2. Search for a path with two x's and no o, and block to keep the player from winning. The loop in rule 1 is executed for rule 2 as well.

3. Search for a square common to at least two paths, of which each contains only an o. If the square common to both paths is moved into, creating a fork, this presents the player with the impossible requirement of blocking two paths at once.

4. Search for a square that would present the machine with a fork if the player moved into it. The search in rule 4 differs from that in rule 3 only in respect to the pathsum of interest. In each case, the

subroutine looks at each cell and all the paths through it (stored in wins_thru_cell), keeps a count (integer k) of the number of paths in the desired state (depending on whether the rule being applied is 3 or 4). There is a special check to discard path zero, which indicates the absence of a path. If count (k) is greater than 1, the cell being inspected is the one to move into. Note that the searches in rule 3 and rule 4 are over each cell, while those in rule 1 and rule 2 are over each path.

5. Search the board, in order of the priority of the cells, for an empty cell. The order of priority is: center, corner cells, and other cells.

6. No empty means draw.

The function won takes as input the cell number of the move last made, determines whether the move is by the player or the machine, and returns .true. if the move won the game for the player, .false. otherwise. The cell number of the move last made is then transformed into an x, y pair, used to address the 3-by-3 array, brd, which uses the same storage as the array, board. This procedure allows examination of all cells in the same row or column, because all will have either the same x or y coordinate. For the cells on a diagonal, an equivalent procedure is adopted to locate all the cells on the same diagonal. The function checks both diagonals, and therefore, unless the cell is in the center, a path that does not contain the cell; but this is safe since if there were a win on the other diagonal the function would have found it in a previous call.

The faulty version of the program follows, with explanatory comments.

A Program to Play Tic-Tac-Toe

```
c       A program to play tic-tac-toe.
c         This version has bugs in it

        common /tic_tac_toe/ board(9)
        integer board
        parameter empty=1
        parameter his=2
        parameter mine=3


c         Logical function won determines whether moves are winners or not.


        logical won


*      Defines the characters used to print the board

        character*1 symbol(3) /" ", "x", "o"/
        logical polite
        data polite /.true./

c         Start up.
c         As game opens, polite is true (as initialized in data statement),
c         so the rules of the game are printed and the board
c         is printed out showing the numbering scheme.
c         The variable is set to .false.,
c         and  subsequently the rules
c         and numbered board
c         are not printed out.
```

```
      if (.not.polite) go to 5
      polite = .false.
      print, "Play tic-tac-toe. Type 1-9 to play."
      print
      print, "            1|2|3"
      print, "            4|5|6"
      print, "            7|8|9"
      print

  5   print, " Your move?"

*       The next statement reads the player's move from the terminal,
*       typed in response to query above.

      read(5, 50, end=97, err=8) move
 50   format(v)
      if(move .gt. 0 .and. move .le. 9) go to 6
  8   print, "Invalid input."
      go to 5

c       The value of move has been checked to see if it is
c       on the board--that is, between 1 and 9.
c       The program must
c       now make sure the move is to an empty square.
c       The variables empty, his, and mine
c       are initialized by the block data subprogram.

  6   if (board(move) .eq. empty) go to 17
      if (board(move) .eq. his) print 11, "You", move
      if (board(move) .eq. mine) print 11, "  I", move
 11   format(1x, a3, " have already played ", i1, ".")
      go to 5

c       The player's move is to an
c       empty square, so put it on the board.

 17   board(move) = his

c       See if this move won game for the player.

      if (won(move)) go to 99

c       The player hasn't won yet, and it is
c       the machine's turn, so it is necessary to get
c       a machine move from the subroutine, mover.

      call mover(move)

c       Check move to see if game is drawn.

      if (move .eq. 0) go to 97

c       Game is not drawn, so inform player of machine's move
c       and put move on board.

      print 52, move
 52   format(" My move is ", i1)
      board(move) = mine

c       Check whether machine has just won. (The won function does it.)
```

```
      if (won(move)) go to 101

c         Since there is no winner yet, print the board and continue play.

      print 51, (symbol(board(i)), i = 1, 9)
 51   format(/2(1x,2(1x,a1,1x,1h¦),1x,a1/1x,11(1h-)/)
     & ,1x,2(1x,a1,1x,1h¦),1x,a1/)
      go to 5

 97   print, " Cat's game."
      go to 100
 99   print, " You win!"
      go to 100
 101  print, " I win."
 100  continue

c         Come here at game's end, regardless of outcome.

      print 51, (symbol(board(i)), i = 1, 9)
      continue
      stop
      end

c     This subroutine will figure out the next move for a game of
c     tic-tac-toe.  The strategy involves looking for an offensive
c     move and then looking for a defensive one of the same priority.

      subroutine mover(move)
      common /tic_tac_toe/ board(9)
      integer board
      parameter empty=1
      parameter his=2
      parameter mine=3
      automatic i, j, k, l, m

 *    All possible paths in the game.  Each path has three cells.
 *    Each number represents a cell. The numbers are ordered
 *    to correspond to the paths along which a game can be won.

      integer paths(3,8)
        data paths /1,2,3,   !path 1
     &              4,5,6,   !path 2
     &              7,8,9,   !path 3
     &              1,4,7,   !path 4
     &              2,5,8,   !path 5
     &              3,6,9,   !path 6
     &              1,5,9,   !path 7
     &              3,5,8/   !path 8

c         The numbers of the paths that pass through a given cell.
c         No cell has more than
c         4 paths through it (center).
c         The corner cells each have 3 paths through them.
c         The rest have 2.  0 represents "no path".

       integer paths_thru_cell (4,9)
        data paths_thru_cell /1,4,7,0,  1,5,0,0,  1,6,8,0,   !cells 1,2,3,
     &                       2,4,0,0,  2,5,7,9,  2,6,0,0,   !cells 4,5,6,
     &                       3,4,8,0,  3,5,0,0,  3,6,7,0/   !cells 7,8,9.

 *     Holds the pathsum, or the sum of the weights of the different
 *     states of a cell.

       integer pathsum(8)
```

```
*       weights for the three states of a cell,
*       in order "empty", "his", and "mine".

        integer weight (3) /0,1,4/


 *      Order in which we will choose a cell when using rule 5.

        integer cells(9)
        data cells /5, 1, 3, 7, 9, 2, 4, 6, 8/


*        calculate the pathsums. The variable k is a counter.

        do 10 i = 1, 8          ! for each path
            pathsum(i) = 0
            do 9 j = 1, 3       ! for each cell in a path
                k = board(paths(j,i))
 9              pathsum(i) = pathsum(i) + weight(k)
 10         continue


 *      Find a path with two in a row for me,
 *      and play a third cell to win
 *      (offensive move, rule 1).

        do 20 j = 1, 8
 20         if (pathsum(j) .eq. (weight(mine)*2)) go to 98 ! is pathsum right?


 *      Find a path with two in a row for his, and play a third cell to block
 *      the path (defensive move, rule 2).

        do 25 j = 1, 8
 25         if (pathsum(j) .eq. (weight(his)*2)) go to 98


 *      Try to make two two-in-a-rows for me (offensive move, rule 3).
 *        For each cell, start counting at 0 for each path_thru_cell,
 *        If there is no path, escape for each cell;
 *        count the number of paths through the cell that have
 *        a pathsum of 4. If it finds two or more
 *        such paths through one cell,
 *        the machine
 *        moves into that cell.

        do 40 move = 1, 9
            k = 0
            do 45 l = 1, 4
                if (paths_thru_cell(l, move) .eq. 0) go to 45
                if (pathsum(paths_thru_cell(l,move)).eq.weight(mine))k=k + 1
 45           continue
 40          if (k .gt. 1) go to 100


 *        Try to block two two-in-a-row for player (defensive move, rule 4)

       do 49 move = 1, 9
           do 47 l = 1, 4
               if (paths_thru_cell(l, move) .eq. 0) go to 47
               if (pathsum(paths_thru_cell(l,move)).eq.weight(his))k=k + 1
 47        continue
 49        if (k .gt. 1) go to 100


 *      No offensive or defensive move so just pick a cell (rule 5).
```

```
      do 60 i = 1,9
          move = cells(i)    ! look through cells in order of priority
          if (board(move) .eq. empty) go to 100 ! is cell empty?
 60   continue

*     No move is found so the game is a draw

      move = 0     !0 means "draw" to caller
      go to 100
 98   do 99 i = 1, 3
          move = paths(i, j)
 99       if (board(move) .eq. empty) go to 100
 100  return
      end

      logical function won(pos)
      common /tic_tac_toe/ board(9)
      integer board
      parameter empty=1
      parameter his=2
      parameter mine=3

      integer pos, brd(3,3), x_or_o, x, y
      logical horizontal, vertical, diagonal_1, diagonal_2
      automatic x_or_o, i, x, y, horizontal, vertical, diagonal_1,
     &    diagonal_2
      equivalence (brd, board)

      horizontal = .true. ; vertical = .true.
      x_or_o = board(pos)


*     x is the row
*     y is the column

      x = mod(pos, 3)
      y = (pos-1) / 3 + 1


*     Check horizontal and vertical simultaneously.

      do 10 i = 1, 3
          if (brd(x,i).ne.x_or_o)horizontal=.false.  !found a cell on this
          if (brd(i,y).ne.x_or_o)vertical=.false.    !path that is in
                                                     !different state
 10       continue

*       Is the cell on a diagonal?

      diagonal_1 = x .eq. y      !is cell on left-to-right downward diagonal?
      diagonal_2 = x + y .eq. 4   !is cell on left-to-right upward diagonal?
      if( .not.diagonal_1 .and. .not.diagonal_2) go to 30   !not on diagonal
      do 20 i = 1, 3
          if (brd(i, i) .ne. x_or_o) diagonal_1 = .false.
 20       if (brd(i, 4-i) .ne. x_or_o) diagonal_2 = .false.
      won = horizontal .or. vertical .or. diagonal_1 .or. diagonal_2
      return
 30   won = horizontal .or. vertical
      return
      end

      block data
      common /tic_tac_toe/ board(9)
      parameter empty=1
      parameter his=2
      parameter mine=3
      integer board/9*empty/
      end
```

Script of Debugging Session

The following is a line by line script of a debugging session. The program in the form given above is debugged with the probe command. The exclamation mark (!) indicates input lines, and the unmarked lines are output. Comments in brackets explain features of the probe command; comments in boxes show programmer's thoughts in the course of debugging.

```
!  bug
 Play tic-tac-toe. Type 1-9 to play.

          1|2|3
          -----
          4|5|6
          -----
          7|8|9
  Your move?
!  1
  My move is 1   <----   [ The program is cheating!
                           You have already played 1. ]
      o |   |
    -----------
        |   |
    -----------
        |   |
  Your move?
!  2
  My move is 1   <----   [ It seems stuck on cell 1. ]

      o | x |
    -----------
        |   |
    -----------
        |   |

  Your move?
!  4
  My move is 2

      o | o |
    -----------
      x |   |
    -----------
        |   |
  Your move?
!  3

Error:  subscriptrange condition by
>user_dir_dir>Multics>JRDavis>doc>fug>bug$main_|1315 (line 284)
A subscript value has exceeded array bounds.
system handler for error returns to command level
  r 1301 0.355 28.102 346 level 2, 13

 !  probe      [You invoke the probe command.]

 Condition subscriptrange raised at line 284 of won.

                        [When probe is invoked after an error it tells
                         you what the error is and where it occurred.]

!  source      [This request displays the source at line 284]

           if (brd(x, i) .ne. x_or_o) horizontal = .false.
           !  found a cell on this path that is in different state
!  value brd (x,i)
 subscript 1 outside range (1:3)
 Cannot get address of brd.
```

```
!  v x

     0                  The calculation of x is wrong when pos is 3,6,or 9
!  v pos <------------  it ought to be x = mod(pos -1,3) + 1.
     3
!  v y
     1

!  let x = 3       [ x is set to the correct value ]

!  after 277    [ This breakpoint will cause x to have the correct value ]

   : if x = 0 : let x = 3
 Break set  after line 277 of bug.

!  continue    [ After an error, continue returns to the command level
               [ from which probe was invoked.                          ]

 r 1303 0.729 68.508 876 level 2, 13

                              [ After a subscriptrange error, ]
!  start                      [ start retries the line where  ]
                              [ error occurred.               ]
  My move is 5

      o | o | x
     -----------
      x | o |
     -----------
          |   |
  Your move?
!  8
  My move is 9
  I win.

      o | o | x
     -----------
      x | o |
     -----------
          | x | o

    STOP

 r 1303 0.363 41.725 381


!  probe bug <------  Explicitly say what program to examine.


!  use mover <------  Explicitly indicate subprogram.


!  before $40:if k > 1 : halt <------  Stop when using rule 3.

                                      [ Even though you specify a FORTRAN  ]
 Break set before line 225 of bug.    [ STATEMENT LABEL the probe command  ]
                                      [ gives you information in terms     ]
                                      [ of source line number.             ]
```

```
!  b $49:if k > 1: halt<----  [Stop when using rule 4.]

Break set before line 235 of bug.

!  ..bug     [The program is called using the ".." escape request]
Play tic-tac-toe. Type 1-9 to play.

          1|2|3
          4|5|6  <----  [It printed the rules again.
          7|8|9          This is a bug.]

 Your move?

!  5<----  [I have already played 5.
            It mistakenly thinks it has played 5.]

 Your move?
!  1
   I have already played 1.<----  [Is this the same board as before?
                                   If so, none of my moves will work, so--]
 Your move?
!  8
 You have already played 8.

 Your move?
QUIT          [QUIT]

r 1304 0.324 22.529 410 level 2, 19
!   probe

Condition quit raised at block|154.  [block is a routine called by the read.]

                  [The where request shows:  1) where bug was when it
!  use bug         "called out"; 2) stack level and name of program;
!  where           3) where control returns when execution resumes
                   after a "continue."]

Current line is line 47 of bug.
Using level 11:  main_.
Control at block|154.

!  v board(*)       ["*" may be used to display every element of the array]
board(1)    3
board(2)    3
board(3)    2
board(4)    2
board(5)    3
board(6)    1
board(7)    1
board(8)    2

board(9)    3
!  l board(*) = 1<----  [This IS the board from last game
!  c                     so set every cell to empty.]

r 1310 0.452 54.132 583 level 2, 19
```

```
!  sr

!  5<———— This time the move is accepted.

 Stopped before line 235 of mover.
!  v k
     2
!  v move
     2
!  v board(move)
 board(2)     1
!  v l
```

Program plans to move into cell 2. The cell is empty as a look at the board shows. All four paths through the cell have been examined. The next step is to check paths-thru-cell to ensure correct data, then pathsums of paths through cell 2 and finally the board.

```
     5

!  v paths_thru_cell(*,move)
 paths_thru_cell(1,2)     1
 paths_thru_cell(2,2)     5
 paths_thru_cell(3,2)     0
 paths_thru_cell(4,2)     0
!  v pathsum(1)
 pathsum(1)     0
!  v pathsum(5)
 pathsum(5)     1
!  v board(*)
 board(1)     1
 board(2)     1
 board(3)     1
 board(4)     1
 board(5)     2
 board(6)     1
 board(7)     1
 board(8)     1
 board(9)     1
```

There is no reason why k should be 2. The board, paths, and pathsum are correct--therefore counting must be wrong. Set a break that will halt before counting.

```
!  b 233:if pathsum(paths_thru_cell(l,move)) = weight(2):halt
 Break set before line 233 of bug.
! ps 30
          if (.not.polite) go to 5


!  b: let board(*) = 1<———— Make board empty before each game.

 Break set before line 32 of bug.

!  quit                [ Each time you type 'quit', one level of invocation
                         of the probe command falls away ]
!  q
 r  1314 0.767 66.092 873
!  bug
 Play tic-tac-toe. Type 1-9 to play.
```

```
        1|2|3
        4|5|6
        7|8|9

 Your move?
!  5
 Stopped before line 233 of mover.
!  v move
     1
!  v 1

     3
!  v k  <---  This is a legitimate instance of a path with
     0        one "o" on it.  It is correct to count it.
!  c          Keep going!

 Stopped before line 233 of mover.

!  v move
     2
!  v 1
     2  <---  This one is correct too.
!  c

 Stopped before line 235 of mover.

!  v k
     2  <---  K is 2? But each move so far had only one
!  v move     path through it with one "o".

     2  <---  K is not being reset to 0 for each move.  You set
              the appropriate break to fix.  List all breaks
              to see which can be reset.
!  status
 Break before line 233.
 Break before line 32.
 Break before line 235.
 Break before line 225.
 Break after line 277.
!  r b 233; r b 235
 Break reset before line 233 of bug.
 Break reset before line 235 of bug.
!  b  236:let k = 0
 Break set before line 231 of bug.
!  c

 My move is 2  <---  It persists with the move it had decided to make.

      | o |
   -----------
      | x |
   -----------
      |   |
```

```
  Your move?
!  8
  My move is 3

       | o | o
    -----------
       | x |
    -----------
       | x |
  Your move?
!  1
  My move is 9

     x | o | o
    -----------
       | x |
    -----------
       | x | o
  Your move?
!  6
  My move is 4

     x | o | o
    -----------
     o | x | x
    -----------
       | x | o

  Your move?
!  7
  Cat's game.

     x | o | o
    -----------
     o | x | x
    -----------
     x | x | o

    STOP <---------- [Perhaps all bugs are now fixed?]

 r 1326 1.949 208.147 2833
!  bug
 Play tic-tac-toe. Type 1-9 to play.

           1|2|3
           4|5|6
           7|8|9

  Your move?
!  5

  My move is 5 <---------- [NO]

       |   |
    -----------
       | o |
    -----------
       |   |
```

```
 Your move?

QUIT <------  ... and release the failing invocation of bug.

r 1327 0.331 22.717 260 level 2, 15
!  rl
r 1327 0.035 3.896 62
!  probe bug

!  a $40:if k > 1:(v move; halt) <------  Breaks were reset too soon.
   Symbol 40 not declared                  Replace them.

! use mover             [You must set the pointer to the desired subroutine
                         or it won't be possible to find the label.]

!  a $40:if k > 1 : (v move; halt)
Break set after line 225 of bug.
!  a $49:if k > 1 : (v move ; halt)
Break set after line 235 of bug.
!  use main_

!  b 33          [Set a break to find out why board is printed each time.]

Break set before line 33 of bug.
!  q
r 1330 0.281 39.723 716
!  bug
Stopped before line 33 of main_.

                        [FORTRAN logical variables are printed as "1"b when
!  v polite              they are .true. and "0"b when .false.]

   "1"b

!  sb polite <------  Why didn't polite keep its value from last time?

logical autcmatic
Declared in main_


!  r <------  It is an automatic variable, instead of a 'save'.
              Now that we know, the break is useless.

Break reset before line 33 of bug.
!  c
Play tic-tac-toe. Type 1-9 to play.

         1|2|3
         4|5|6
         7|8|9

 Your move?
!  5
 My move is 5 <------  Why didn't it hit the break?
          |   |
      -----------
          | o |
      -----------
          |   |

 Your move?
```

```
!  1

 Stopped before line 225 of mover.
```

Hit the rule 3 break.

```
!  st
 Break after line 235.
 Break before line 231.
 Break before line 32.
 Break before line 225.
 Break after line 225.
 Break after line 277.
```

The list of breaks shows breaks both before and after line 225, and after 235. The breaks ought to be before, because breaks after a line aren't executed if the line does a goto.

```
!  st at 235
 Break after line 235:if k > 1 : (v move ; halt)
!  r a 235
 Break reset after line 235 of bug.
!  b 235:if k > 1: (v move ; halt)
 Break set before line 235 of bug.
!  q
 r 1335 0.506 45.840 592
!  bug
 Play tic-tac-toe. Type 1-9 to play.

           1|2|3
           4|5|6
           7|8|9

  Your move?
!  5
    5

 Stopped before line 235 of mover.
```

This time it hit the break.

```
!  v k
    4
!  v board(*)
 board(1)    1
 board(2)    1
 board(3)    1
 board(4)    1
 board(5)    2
 board(6)    1
 board(7)    1
 board(8)    1
 board(9)    1
!  v paths_thru_cell(*,5)
 paths_thru_cell(1,5)    2
 paths_thru_cell(2,5)    5
 paths_thru_cell(3,5)    7
 paths_thru_cell(4,5)    8


!  v pathsum(*)
 pathsum(1)    0
 pathsum(2)    1
 pathsum(3)    0
 pathsum(4)    0
 pathsum(5)    1
 pathsum(6)    0
 pathsum(7)    1
 pathsum(8)    1
!  st

 Break before line 235.
 Break before line 231.
 Break before line 32.
 Break before line 225.
```

The board, paths, and pathsums are correct. You realize that the code doesn't check to see if the cell is empty. Rule 4 is stated as 'Find a cell that is the intersection of two paths, each of which has one "o"; it should be 'Find an empty cell that is....' You can't patch this bug with a probe break, so you must edit the source and recompile. List all the breaks, as a reminder of where the source needs correction.

```
 Break after line 277.
 Break after line 225.
!  st b 32
 Break before line 32:let board(*) = 1
!  st a 277
 Break after line 277:if x = 0 : let x = 3
!  q
 r 1337 0.751 51.764 736


!  two<------- The edited program is now named 'two'.

 Play tic-tac-toe. Type 1-9 to play.

         1¦2¦3
         4¦5¦6
         7¦8¦9

  Your move?
!  5
  My move is 1

       o ¦   ¦
      -----------
         ¦ x ¦
      -----------
         ¦   ¦
  Your move
!  3

  My move is 8 <------- Should be 7. Rule 2 is not working.
       o ¦   ¦ x
      -----------
         ¦ x ¦
      -----------
         ¦ o ¦
  Your move?
!  7
  You win!

       o ¦   ¦ x
      -----------
         ¦ x ¦
      -----------
       x ¦ o ¦

!  probe two

!  use mover
!  b $98<------- Set a break at the beginning of the loop
                 transferred to by rule 2.

Break set before line 259 of two.
!  q
 r 1422 0.152 29.289 387
!  two
  Your move?
!  5
  My move is 1

       o ¦   ¦
      -----------
         ¦ x ¦
      -----------
         ¦   ¦
```

```
 Your move?

Stopped before line 259 of mover.

!  v j <---- What path is it trying to block you on?

    8
!  v pathsum(8)
pathsum(8)    2
!  v board(*)
board(1)    3
board(2)    1
board(3)    2
board(4)    1
board(5)    2
board(6)    1
board(7)    1

board(8)    1
board(9)    1 <---- Path 8 is right, as is pathsum and board.

!  v paths (*,8)

paths(1,8)    3
paths(2,8)    5
paths(3,8)    8 <---- The path was not defined right. Must have
                      made a typing error.

!  l paths(3,8) = 7

!  q
r 1428 0.057 10.033 105
!  two
 Your move?
!  5
 My move is 1

     o ¦   ¦
    -----------
       ¦ x ¦
    -----------
       ¦   ¦
 Your move?
!  3
Stopped before line 259 of mover.
!  r b 259
Break reset before line 259 of two.
!  c
 My move is 7

     o ¦   ¦ x
    -----------
       ¦ x ¦
    -----------
     o ¦   ¦
 Your move?
!  6
 My move is 4
 I win.

     o ¦   ¦ x
    -----------
     o ¦ x ¦ x
    -----------
     o ¦   ¦
```

```
    STOP

 r 1429 0.309 35.534 512
!  two

  Your move?  <---- Maybe all bugs are fixed?
!  2
  My move is 5

       | x |
   -----------
       | o |
   -----------
       |   |
  Your move?
!  4
  My move is 1

     o | x |
   -----------
     x | o |
   -----------
       |   |
  Your move?
!  9
  My move is 3

     o | x | o
   -----------
     x | o |
   -----------
       |   | x
  Your move?
!  7
  My move is 8

     o | x | o
   -----------
     x | o |
   -----------
     x | o | x
  Your move?
!  6
  Cat's game.

     o | x | o
   -----------
     x | o | x
   -----------
     x | o | x

    STOP

r 1429 0.324 18.500 333

!  two  <---- One more test case

  Your move?
!  1
  My move is 5

     x |   |
   -----------
       | o |
   -----------
       |   |
  Your move?
```

```
 !  9
   My move is 3

       x ¦   ¦ o
     -----------
         ¦ o ¦
     -----------
         ¦   ¦ x
   Your move?
 !  7
   My move is 8

       x ¦   ¦ o
     -----------
         ¦ o ¦
     -----------
       x ¦ o ¦ x
   Your move?
 !  4
   You win!

       x ¦   ¦ o
     -----------
       x ¦ o ¦
     -----------
       x ¦ o ¦ x

     STOP
```

The algorithm itself is defective. Correction is left to the interested reader. Hint: the blocking strategy used by rule 4 is the incorrect part.

Corrected Program to Play Tic-Tac-Toe

```
c       A program to play tic-tac-toe.

        common /tic_tac_toe/ board(9)
        integer board
        parameter empty=1
        parameter his=2
        parameter mine=3

        logical won
*      defines the characters used to print the board
        character*1 symbol(3) /" ", "x", "o"/
       integer i
        logical polite
        save polite
        data polite /.true./

c       Start up.

*      clear the board for a new game
        do 2 i = 1,9
 2         board(i) = empty
```

```
        if (.not.polite) go to 5
        polite = .false.
        print, "Play tic-tac-toe. Type 1-9 to play."
        print
        print, "            1¦2¦3"
        print, "            4¦5¦6"
        print, "            7¦8¦9"
        print

  5     print, " Your move?"
        read(5, 50, end=97, err=8) move
  50    format(v)
        if(move .gt. 0 .and. move .le. 9) go to 6
  8     print, "Invalid input."
        go to 5

c            move has been checked it is on the board.
c            make sure it is an empty square
  6     if (board(move) .eq. empty) go to 17
        if (board(move) .eq. his) print 11, "You", move
        if (board(move) .eq. mine) print 11, "  I", move
  11    format(1x, a3, " have already played ", i1, ".")
        go to 5

c         move is to an empty square, so put it in the board
  17    board(move) = his
c         see if user has won
        if (won(move)) go to 99
c         user hasnt won, get a machine move
        call mover(move)
c         see if game is really drawn
        if (move .eq. 0) go to 97
c         game isnt drawn, so tell user what machine did, and put move in
        print 52, move
  52    format(" My move is ", i1)
        board(move) = mine
c         see if machine has just won
        if (won(move)) go to 101
c            no winner yet, print board and continue to play
        print 51, (symbol(board(i)), i = 1, 9)
  51    format(/2(1x,2(1x,a1,1x,1h¦),1x,a1/1x,11(1h-)/)
       &  ,1x,2(1x,a1,1x,1h¦),1x,a1/)
        go to 5
  97    print, " Cat's game."
        go to 100
  99    print, " You win!"
        go to 100
  101   print, " I win."
  100   continue

c            come here at games end, regardless of outcome
        print 51, (symbol(board(i)), i = 1, 9)
        continue
        stop
        end

c       This subroutine will figure out the next move for a game of
c       tic-tac-toe.  The strategy involves looking for an offensive
c       move and then looking for a defensive one of the same priority.

        subroutine mover(move)
        common /tic_tac_toe/ board(9)
        integer board
        parameter empty=1
        parameter his=2
        parameter mine=3

       automatic i, j, k, l, m
```

```
*       All possible paths in the game.
        integer paths(3,8)
          data paths /1,2,3, !path 1
     &               4,5,6, !path 2
     &               7,8,9, !path 3
     &               1,4,7, !path 4
     &               2,5,8, !path 5
     &               3,6,9, !path 6
     &               1,5,9, !path 7
     &               3,5,7/ !path 8

*       The numbers  of paths which pass through a given cell.
        integer paths_thru_cell (4,9)
        data paths_thru_cell /1,4,7,0, 1,5,0,0, 1,6,8,0, 2,4,0,0,
       &                      2,4,0,0, 2,5,7,8, 2,6,0,0,
       &                      3,4,8,0, 3,5,0,0, 3,6,7,0

*       Holds the pathsum
        integer pathsum(8)

*       weights for the three states of a cell
        integer weight (3) /0,1,4/

*       Order in which we will choose a cell. (rule 5)
        integer cells(9)
        data cells /5, 1, 3, 7, 9, 2, 4, 6, 8/

*       calculate the pathsums
        do 10 i = 1, 8
            pathsum(i) = 0
            do 9 j = 1, 3
                k = board(paths(j,i))
 9              pathsum(i) = pathsum(i) + weight(k)
 10         continue

*       Find two in a row for me, and win.

        do 20 j = 1, 8
 20         if (pathsum(j) .eq. (weight(mine)*2)) go to 98

*       Find two in a row for his, and block it.

        do 25 j = 1, 8
 25         if (pathsum(j) .eq. (weight(his)*2)) go to 98

*       Try to make two two-in-a-rows for me (offensive).

        do 40 move = 1, 9
            k = 0
            if (board(move) .ne. empty) go to 40
            do 45 l = 1, 4
                if (paths_thru_cell(l, move) .eq. 0) go to 45
                if (pathsum(paths_thru_cell(l,move)).eq.weight(mine)) k = k + 1
 45           continue
 40         if (k .gt. 1) go to 100

*         try to block two two-in-a-row for user (defensive)

       do 49 move = 1, 9
            k = 0
            if (board(move) .ne. empty) go to 49
            do 47 l = 1, 4
               if (paths_thru_cell(l, move) .eq. 0) go to 47
               if (pathsum(paths_thru_cell(l,move)).eq.weight(his)) k = k + 1
 47         continue
 49         if (k .gt. 1) go to 100

*       No offensive or defensive move so just pick a cell.
```

```
      do 60 i = 1,9
          move = cells(i)
          if (board(move) .eq. empty) go to 100
 60   continue

*     no move is found so the game is a draw
      move = 0
      go to 100
 98   do 99 i = 1, 3
          move = paths(i, j)
 99       if (board(move) .eq. empty) go to 100
 100  return
      end
      logical function won(pos)
      common /tic_tac_toe/ board(9)
      integer board
      parameter empty=1
      parameter his=2
      parameter mine=3

      integer pos, brd(3,3), x_or_o, x, y
      logical horizontal, vertical, diagonal_1, diagonal_2
      automatic x_or_o, i, x, y, horizontal, vertical, diagonal_1,
     &    diagonal_2
      equivalence (brd, board)

      horizontal = .true. ; vertical = .true.
      x_or_o = board(pos)

*     x is the row
*     y is the column

      x = mod(pos -1,3) + 1
      y = (pos-1) / 3 + 1

*     Check horizontal and vertical simultaneously.

      do 10 i = 1, 3
          if (brd(x, i) .ne. x_or_o) horizontal = .false.
          if (brd(i, y) .ne. x_or_o) vertical = .false.
 10       continue

*     Check diagonal if possible.

      diagonal_1 = x .eq. y
      diagonal_2 = x + y .eq. 4
      if( .not.diagonal_1 .and. .not.diagonal_2) go to 30
      do 20 i = 1, 3
          if (brd(i, i) .ne. x_or_o) diagonal_1 = .false.
 20       if (brd(i, 4-i) .ne. x_or_o) diagonal_2 = .false.
      won = horizontal .or. vertical .or. diagonal_1 .or. diagonal_2
      return
 30   won = horizontal .or. vertical
      return
      end
```

APPENDIX B

OPTIMIZATION

This chapter describes many of the optimizations performed by the Multics FORTRAN compiler when you specify the -optimize or -safe_optimize control arguments. The information in this section is of primary interest to advanced users.

LOCAL OPTIMIZATIONS

Local optimizations are those that improve the code generated for a particular statement without considering or affecting the rest of the program. "Improved code" is code that is executed more efficiently, or faster, than code that is not optimized.

Machine-Independent Local Optimizations

IMPROVEMENT OF LOGICAL if STATEMENTS

This optimization improves the code for a logical if statement so that expressions not needed in determining the outcome of the statement are not evaluated. This optimization makes the more readable

```
     if(m .lt. 4 .and. n .gt. 5) i=3
```

as efficient as

```
     if (m .ge. 4) goto 100
     if (n .le. 5) goto 100
     i=3
100  continue
```

Note that in this case, function references contained in logical expressions in a logical if statement are not necessarily evaluated. Programs that depend on the evaluation of the entire

expression in a logical if statement are in error, and may not execute as desired if you compile them with the -optimize control argument.

Machine-Dependent Local Optimizations

The following machine or implementation dependent optimizations are of special interest.

QUICK SUBPROGRAM CALL

If subprogram A and subprogram B are compiled seperately and A calls B, a full Multics calling sequence is generated, executing many instructions and requiring the creation of a Multics stack frame for both subprograms. If subprograms A and B are compiled together (in the same invocation of the fortran command), the compiler optimizes the call from A to B. Thus,

```
subroutine a
    .
    .
    .
call b
    .
    .
    .
end
    .
    .
    .
subroutine b
    .
    .
    .
end
```

is much more efficient than

```
subroutine a           subroutine b
    .                      .
    .                      .
    .                      .
call b                 end
    .
    .
    .
end
```

See Sections 3 and 5 for discussions of these issues from a different viewpoint.

IMPLIED DO-LOOPS

Doing I/O on an array using a reference to the array as a whole is normally more efficient than doing I/O on an array using an implied do-loop with an array element, because the former executes one call to the I/O system for the entire array, for every data transfer, while the latter executes one call to the I/O system for every array element. However, the compiler attempts to recognize implied do-loops that do I/O on a vector (a contiguous area of storage) and replaces these loops with I/O calls for the entire vector. Thus the data transfer statement

```
write(6)(a(i), i=1,100)
```

with this optimization can become as efficient as

```
write (6) a
```

assuming that "a" is an array of 100 elements. The optimizing algorithm processes nested implied do-loops from inside out. A partial list follows of the constraints that must be satisfied at each level in order for this optimization to take place:

- The item to be transmitted must be subscripted
- The subscript at a particular level must be the same as the index of the do at the same level
- The increment of the do-loop must be 1
- The array element must be the only item in the data list of the implied do-loop at this level
- The subscript at a particular level must not be a potential alias of any other subscript
- Inner implied do-loops must cover a complete dimension

Since in FORTRAN an array is allocated so that leftmost subscripts vary most rapidly, the inner loops should match the innermost dimension. Thus, given

```
dimension a(20,30)
```

the optimization occurs for

```
write (6) ((a(i,j), i=1,20), j=1,30)
```

but not for

```
write (6) ((a(i,j), j=1,30), i=1,20)
```

GLOBAL OPTIMIZATIONS

Global optimizations are those that optimize code over more than one statement and/or consider conditions over an entire program unit. This type of optimization is performed for every program unit in a compilation if you invoke the compiler with the -optimize or -safe_optimize control argument.

Machine-Independent Global Optimizations

Most of the global optimizations done by the Multics FORTRAN optimizer are machine-independent, except that it is assumed that an addition executes faster than a multiplication does. These global optimizations can be classified according to whether they are meant to improve the execution of a loop or are more general in their effect. The more general optimizations are discussed first in what follows.

Non-Loop-Oriented Optimization

REMOVAL OF COMMON SUBEXPRESSIONS

This optimization attempts to avoid repeated evaluation of the same expression. When the expression first occurs, its value is saved in a temporary compiler-created variable, and that value is used when the expression occurs again instead of its being reevaluated. The optimization occurs for two instances of the same expression only if all the following constraints are satisfied.

- the first instance of the expression must always be executed before the second
- none of the input operand values can change between the two instances
- all operators in the expression, including functions, yield a given result with given operands (e.g., 2+2 is always 4, not sometimes 4 and sometimes 3)
- no operators in the expression have "side-effects." A side-effect means doing I/O or changing the value of a variable other than by means of the assignment operator.

In the following case

```
a + b*c
   .
   .
   .
a + b*c
```

the value of a + b*c is calculated only once and is used in all other instances of that expression as long as the above constraints are satisfied. Note that in the case

```
b + c
  .
  .
  .
a + b + c
```

there is no common expression to eliminate because the rules of FORTRAN require the second expression to be seen as (a + b) + c. If you write

```
b + c
  .
  .
  .
a + (b + c)
```

The expression b + c becomes common and is evaluated only once.

CONSTANT PROPAGATION

This optimization causes expressions with constant operands to be evaluated at compile time, and causes the known constant values of variables to be propagated to where the variable occurs in other expressions. This latter case might result in the recognition of other constant expressions. For example,

```
i = 3*4 gives i = 12
j = i+5 gives j = 17
```

This optimization is not applied to arrays or to array elements. Most built-in functions with constant arguments are not, as of MR7.0, evaluated at compilation time.

LOOP-ORIENTED OPTIMIZATIONS

All loop-oriented optimizations are made under the assumption that code inside a loop is executed more often than code outside the loop. These optimizations, therefore, try to replace computations inside a loop with other computations outside the loop. Wherever the basic assumption does not hold, the optimizer will not succeed in speeding up execution.

REMOVAL OF INVARIANT EXPRESSIONS FROM LOOPS

This is the most obvious loop-oriented optimization. If an expression is loop-invariant, it is evaluated ahead of the loop itself, and its value is then used inside the loop. In order for this optimization to be performed on a particular expression, the following constraints must be satisfied:

- all input operands must not change their values within the loop
- all operators must produce the same value for the same set of inputs
- if the -safe_optimize control argument is specified, or if the operation is likely to cause an interrupt to occur if given bad inputs (e.g., /,**,sqrt, asin), then the expression must be in a part of the loop that is always executed if the loop is entered.

For example, take the following loop:

```
     do 100 i=1,10000
     array(i) = i+x**y
100  continue
```

This would be transformed into

```
     T = x**y
     do 100 i = 1,10000
     array(i) = i+T
100  continue
```

where T is a temporary, with a substantial decrease in execution time.

Assignment statements may also be removed from loops. To remove an assignment statement from a loop, the above mentioned constraints must hold for the expression on the right hand side of the equal sign. In addition, the following constraints must be satisfied:

- The target of the assignment must not be referenced in the loop before the assignment statement is referenced
- The value of the target of the assignment must not be changed elsewhere in the loop
- The assignment statement must be in a part of the loop that is always executed if the loop is entered.

For example, the following loop

```
      do 100 i = 1,10000
      a = x**y
      array(i) = a*i
100   continue
```

is transformed by the optimizer to

```
      a = x**y
      do 100 i = 1,10000
      array(i) = a*i
100   continue
```

with no change in the meaning of the program, but the following loop is not transformed because it violates the above constraints and would change the meaning of the program:

```
      do 100 i = 1,10000,2
      a = x**y
      array(i) = a*i
      a = x/y
      array(i+1) = a*i
100   continue
```

In this case, the assignment statements setting "a" could not be removed, although x**y and x/y could be removed.

-safe_optimize VS. -optimize

It is a goal of a good optimizer that a valid program should produce the same results given the same inputs, whether optimized or not. Obviously, if optimization causes a fault to occur when an optimized program is run, and running the unoptimized program does not cause a fault, this goal is not met. This unfortunate situation could occur if the optimizer removed an expression from a part of the loop that is not always executed if the loop is entered. For example, removing x/y from the following loop would be a mistake:

```
      do 100 i = 1,10000
      array(i) = i
      if (y .ne. 0) array(i) = (x/y)*i
```

```
      100 continue
```

because a zerodivide fault would occur if y were equal to zero. Therefore operations such as /, **, and most built-in function references are not removed from loops unless they are in the part of the loop that is always executed, whether or not the -safe_optimize control argument is specified. The operators +, -, and * present a different case. They are extremely unlikely to cause a fault, no matter what inputs they receive. Therefore, if the -optimize control argument is specified, these operators may be removed from a loop even if not in the part of the loop that is always executed. This raises a very slight possibility that a fixedoverflow, overflow, or underflow fault might occur with the optimized program that would not occur with the unoptimized program. For this rare case, the -safe_optimize control argument is available. If the -safe_optimize control argument is specified, the "always-executed" constraint is rigidly enforced for the +, -, and * operators. It is recommended that you not specify the -safe_optimize control argument unless you really have to, since it inhibits much valuable optimization and the likelihood that it is needed is extremely low. (A better way to get the effect of specifying the -safe_optimize control argument is to specify the safe keyword in a %options or %global statement. This statement has the added advantage of making it explicit in the source that the "always-executed" constraint must be rigidly enforced. Again this is not recommended unless testing shows it to be necessary.)

STRENGTH REDUCTION

An <u>induction variable</u> is one that is altered within a loop only by incrementing it by a constant or loop-invariant variable. The index variable of a do-loop is an example of an induction variable. The optimizer also recognizes induction variables that are altered by assigning to them a simple linear function of another induction variable and loop-invariant expressions or constants. Induction variables are frequently multiplied by or added to constants or loop-invariant expressions, and these operations can be "reduced" to cheaper additions by introducing new induction variables. This form of optimization is called strength reduction.

For example, consider the program fragment:

```
           common a(50,50)
           do 100 i = 1,500
           a(1,i) = a(1,i) + 3.0
      100 continue
```

In its internal representation, the compiler sees this as equivalent to:

```
        i = 1
10      a(50*i) = a(50*i)+ 3.0
        i = i + 1
        if (i .le.  500) goto 10
```

To replace 50*i by a new induction variable TI, we initialize TI to 50*i before the loop and increment TI by 50 after i is incremented:

```
        i = 1
        TI = 50*i
10      a(TI) = a(TI) + 3.0
        i = i + 1
        TI = TI + 50
        if (i .le.  500) goto 10
```

The resulting program fragment is faster because of the eliminated multiplications. Further optimizations will speed up the fragment even further.

TEST REPLACEMENT

It is quite likely that strength reduction will have eliminated from a loop all references to an induction variable except the incrementing of the variable and, possibly, a test of that variable against a loop constant to exit the loop. If this is true, and if the value of the induction variable is not needed after the loop is executed, the incrementing of the induction variable is deleted and the test, if any, is changed to use one of the newly created induction variables.

For example, if we take the program fragment described in the strength reduction section and apply test replacement to it, we get:

```
        i = 1
        TI = 50*i
10      a(TI) = a(TI) + 3.0
        TI = TI + 50
        if (TI .le.  25000) goto 10
```

REMOVAL OF DEAD ASSIGNMENTS

The optimizer applies its algorithms to loops from the inside out. If we take the program fragment described above, and apply constant propagation to the outer loop, we get:

```
          i = 1
          TI = 50
     10   a(TI) = a(TI) + 3.0
          TI = TI + 50
          if (TI .le.  25000) goto 10
```

Notice that this has made the assignment to "i" useless since its value is never used. This is called a "dead assignment." The optimizer then removes this dead assignment giving:

```
          TI = 50
     10   a(TI) = a(TI) + 3.0
          TI = TI + 50
          if (TI .le.  25000) goto 10
```

for a tremendous improvement in efficiency over the original program fragment.

At present, the optimizer removes only those assignments made dead by a combination of test replacement and constant propagation.

Machine-Dependent Global Optimization

As of MR10.2, several machine-dependent global optimizations are performed by the compiler.

GLOBAL POINTER REGISTER USE

Pointer register management attempts optimal use of pointer registers allocated across loops. During the first pass of the optimizing code generator, an analysis of potential pointer register use is made, usage counts are determined, and the most frequently used global pointers are set up to be loaded at the start of loops, scanning from the innermost to the outermost loops.

GLOBAL INDEX REGISTER

Index register management attempts optimal use of index registers allocated across loops. During the first pass of the optimizing code generator, an analysis of potential index register use is made, usage counts are determined, and the most frequently used global indexes are set up to be loaded at the start of loops, scanning from the innermost to the outermost.

PROCESSOR INSTRUCTION FETCH PADDING

Both the optimizing and nonoptimizing code generators pad the entry label within do-loops in order to start them on an even-word boundary. This optimization causes the processor to have always two executable instructions available from the fetch at the start of the loop, which produces as much as an eight percent improvement over starting at an odd-word boundary. The optimizing code generator does this only for the innermost loops in order to reduce code size, while the nonoptimizing compiler does this to all loops.

This page intentionally left blank.

By applying the optimization described here to the program segment used in the previous section, the following object code results:

```
         epp7     pr4/q,*       addr (common block)
         lxl2       50,dl

$10:     fld      pr7/-50,2     a
         fad      =3.0,du
         fstr     pr7/-50,2     a

         adlx2    50,du
         cmpx2    25000,du
         tmoy     $10
```

POINTERS FOR EFFICIENT CODING STYLE USING THE FORTRAN OPTIMIZER

- When using arrays in loops, have the leftmost subscripts vary most rapidly. This decreases paging.

- If a set of subprograms is called during the execution of a program, compile the subprograms and the main program together in the same source segment. This method of compilation produces faster calling sequences, as described above in the beginning of this section under "Quick Subprogram Call."

- If no arguments in a quick call to a subroutine or function are subscripted, or are formal parameters or are in common storage, the calling sequence will be shorter.

- Avoid the use of equivalenced variables and formal parameters as do-loop indexes and array subscripts. Because of potential aliasing, the optimization of expressions containing these variables is partially inhibited.

- Avoid the use of variables in common as do-loop indexes and as subscripts in loops that contain subroutine calls, function references, or references to formal parameters.

- Code programs in a straight-line manner and avoid circuitous and involved logic with many goto statements. Use do-loops for looping rather than if statements. Although the optimizer recognizes all loops whether or not they are do-loops, optimization is often better with do-loops because, topologically, they tend to be well-formed loops.

- Avoid extended range do-loops.

- Avoid use of assigned goto statements. Besides making a program more obscure to the reader, assigned goto statements are not handled well because they obscure the program flow.

- Avoid end= or err= branches to points within loops. These branches are best used to points outside of all loops in a program unit.

- Control all do-loops with integer variables rather than real or complex variables. Strength reduction is applied only to integer variables.

- Use a save statement to name local variables whose values must be saved across successive invocations of a program unit. Otherwise, do not use a save statement. Use automatic variables when values need not be saved.

- Order the terms in a complicated logical expression in a logical if statement such that the most likely relational expression to decide the result comes first.

- Attempt to do I/O on whole arrays or vectors rather than scattered array elements when inputting or outputting arrays.

APPENDIX C

COMPATIBILITY WITH NON-FORTRAN PROGRAMS

FORTRAN programs can reference PL/I procedures exactly as they would reference subroutine or function subprograms. There are no restrictions on the types of arguments passed from a FORTRAN subprogram to a PL/I procedure. The following table indicates the proper PL/I declaration for each FORTRAN data type.

WARNING: As a result of differences between PL/I and FORTRAN languages, bit(1) aligned in PL/I is not equivalent to logical in FORTRAN.

FORTRAN	PL/I
integer i,j,k	declare (i,j,k) fixed bin(35);
real a,b	declare (a,b) float bin;
double precision d,e	declare (d,e) float bin(63);
complex c	declare c float bin complex;
logical r,s	declare (r,s) bit(36) aligned;
character*7 cs	declare cs char(7) aligned;
call subr (x,y,z,$10,$5)	integer_value=0; call subr (x,y,z,integer_value); /*if integer_value>0 and <=number of label_args, then go to label_array (integer_value)*/
subroutine subr (a,b,c,*)	subr: procedure(a,b,c)returns (fixed bin);
return 5	return (5);

FORTRAN subprograms may be called by PL/I procedures and may receive arguments of any FORTRAN data type.

Multidimensional PL/I arrays can be passed to FORTRAN and vice versa if the dimensions are reversed <u>and</u> the subscripts are reversed. This is because FORTRAN stores arrays in column-major order while PL/I stores them in row-major order.

Example:

```
dimension q(5,10),r(10)          x:   proc(a,b,c);
real q,r,s                       dcl   a(10,5) float,
     .                           b(10) float,
     .                           c float;
     .                                .
call x(q,r,s)                         .
     .                                .
     .                                .
     .
     .
```

Note: q(2,3) is referenced in x as a(3,2).

The FORTRAN call is identical to a PL/I call and is fully compatible with Multics standards. FORTRAN subprograms can call and be called by programs written in any language that obeys these conventions and implements Multics standard data types that correspond to the FORTRAN data types. FORTRAN subprograms may call any of the Multics system entries whose arguments are restricted to the data types available in Multics FORTRAN.

Three classes of procedure require descriptors in PL/I: (1) all procedures declared "options (variable)," (2) all procedures declared with star extents, and (3) all Multics commands. A PL/I procedure is considered to have star extents if its declaration (or Usage Description) contains the character "*".

If a called PL/I procedure expects descriptors, the calling FORTRAN program must declare the entry name in an external statement with the "(descriptors)" attribute.

Example:

```
external ioa_(descriptors), sort_seg (descriptors)
     .
     .
     .
call ioa_("Error retype the values")
     .
     .
     .
call ioa_("Job ^a,^d lines completed", job_name, line_count)
     .
     .
call sort_seg ("temp_file")
```

Example:

PL/I	FORTRAN
dcl a entry(fixed bin, char (5)); dcl b entry(fixed bin, char (*));	external b (descriptors) . . . call a (5, "Hello") . . . call b (-4, "Bye")

In the example above the declarations on the left are PL/I and the code on the right shows how the declaration is written in FORTRAN. Notice that b requires descriptors; a does not.

APPENDIX D

ERROR MESSAGES

A complete list follows of compilation-time error messages and runtime I/O error messages. In the case of the compilation-time messages, xxx indicates the point in the message where specific information about the actual error is inserted by the compiler. The runtime I/O messages would also be more specific in real cases.

All error messages that begin with "Compiler error:" are errors in the compiler itself and should never occur. If such an error message does occur, you should report it to maintenance personnel at your site, and save the source so that the error can be duplicated and corrected by the developers.

COMPILE-TIME ERROR MESSAGES

ERROR NUMBER	SEVERITY	MESSAGE
001	3	This segment contains no FORTRAN statements, just comment lines.
002	3	Extra end statement encountered.
003	3	Main program must be the first program unit in the segment.
004	2	Executable statements cannot appear in a block data subprogram.
005	1	This statement is preceded by an unconditional transfer of control and cannot be referenced.
006	3	Syntax error in xxx statement. Text follows logical end of statement.
007	2	This subprogram must contain at least one

executable statement.

008	3	The do loop ending with xxx has not been terminated.
009	2	Return value of function xxx has not been set.
010	3	Syntax error. A variable name is required in place of xxx.
011	3	Syntax error. A right parenthesis is required in place of xxx.
012	3	The label xxx has been referenced but not declared.
013	3	Syntax error. A slash is required in place of xxx.
014	3	xxx is declared with variable bounds but is not a parameter.
015	1	More than one unnamed block data subprogram.
016	3	A xxx statement cannot terminate a do loop.
017	2	A xxx statement cannot appear in a main program.
018	3	xxx is the index of two or more nested implied do loops.
019	3	xxx, which is a bound of xxx, is not an scalar integer, parameter, constant, or in_common.
020	3	xxx cannot be declared as a member of the common block xxx.
021	3	xxx cannot be referenced as a subroutine.
022	3	Syntax error. A left parenthesis is required in place of xxx.
023	3	Syntax error. A statement label is required in place of xxx.
024	3	Syntax error. An unsigned integer constant is required in place of xxx.

025	3	Syntax error. A reference to a scalar variable is required in place of xxx.
026	3	Syntax error. A comma is required in place of xxx.
027	3	Implementation restriction: Do loop and block if nesting has exceeded xxx.
028	3	Syntax error. An equivalence group must contain at least two members.
029	2	An xxx statement file must be a variable, array, or array element of arithmetic or character type.
030	3	The xxx attribute is redundant or conflicting for xxx and is ignored.
031	3	A xxx statement must have a xxx specification.
032	3	This is not an assignment statement and xxx is not a known keyword.
033	3	A global save statement must be the only save statement in a program unit.
034	3	Syntax error. A left parenthesis encountered that does not delimit an implied do loop.
035	3	A xxx statement cannot appear in a main program.
036	1	The real constant xxx has more than xxx digits and has been converted to double precision.
037	3	Syntax error. A format statement must have a statement label.
038	3	A program unit cannot contain both automatic statements and save statements.
039	3	xxx can only appear in this parameter list once.
040	3	Syntax error. A keyword is required in place of xxx.
041	3	Syntax error. A label, variable name, or list of labels is required in place of xxx.
042	3	A xxx statement cannot be the second part of a logical if statement.
043	1	xxx has been referenced but not set.

044	3	Syntax error. The keyword xxx is required in place of xxx.
045	3	Syntax error. A single letter is required in place of xxx.
046	3	Syntax error. The letter range specified must be in ascending alphabetical order.
047	3	Syntax error. The letters in the letter range must be in the same case.
048	3	The xxx keyword has been specified more than once in this statement.
049	3	Syntax error. An equals sign is required in place of xxx.
050	3	Syntax error. A constant is required in place of xxx.
051	3	Syntax error. Only arithmetic constants can be signed.
052	3	Syntax error. The characters xxx are out of place.
053	3	Syntax error. A character string constant is required in place of xxx.
054	3	Pathname in library statement is not acceptable.
055	3	Implementation restriction: A statement function is limited to xxx arguments.
056	3	This xxx statement is out of sequence and is ignored.
057	1	Adding a word to common block xxx in order to store xxx on a double word boundary.
058	3	xxx is a member of common and cannot have variable bounds.
059	3	Storage class for xxx conflicts with storage class of the equivalence group.
060	3	Attempt to equivalence xxx to more than one location.
061	3	Attempt to change the address of common block xxx.
062	3	Cannot equivalence a member of a common block, xxx, to another common block, xxx.

063	3	Alignment requirements for equivalence group containing xxx cannot be resolved.
064	3	xxx cannot appear in an equivalence group.
065	3	xxx cannot appear in an equivalence group because it has variable bounds.
066	3	Syntax error. xxx is not a valid keyword for the xxx statement.
067	4	Implementation restriction: xxx has overflowed its limit of xxx words.
068	1	xxx, which is a formal parameter of the statement function xxx, has not been referenced.
069	2	The statement label xxx is less than 1 or greater than 99999.
070	3	The statement label xxx has been previously defined. This definition is ignored.
071	3	This executable statement label is used as a format specification.
072	3	This format statement label is used in an executable context.
073	3	The statement label on this statement cannot be referenced.
074	3	A reference to an executable statement label is required in place of xxx.
075	3	A reference to a format statement label is required in place of xxx.
076	3	A subscripted reference to xxx is not possible.
077	3	This reference to xxx is not valid because it has variable bounds.
078	3	The subscript xxx exceeds the corresponding xxx bound for xxx.
079	3	A reference to xxx has xxx subscripts.

080	3	Initialization of common blocks can only occur in a block data subprogram. xxx not initialized.
081	2	More than one initial value assigned to an element of xxx.
082	3	Fewer constants than variables in a data specification.
083	3	The mode of xxx is not compatible with the mode of xxx.
084	3	More constants than variables in a data specification.
085	3	Implementation restriction: Format specification is longer than xxx characters.
086	3	Syntax error in a format specification. xxx
087	3	Implementation restriction: The character length of xxx exceeds xxx.
088	3	Syntax error. A format or namelist reference is required in place of xxx.
089	3	Implementation restriction: Implied do loop nesting exceeds xxx.
090	3	Syntax error. Parentheses do not balance.
091	3	Syntax error in an implied do loop.
092	3	xxx is not a keyword or variable name and cannot start a FORTRAN statement.
093	3	xxx cannot be declared as a builtin function.
094	3	Syntax error. An operand is required in place of xxx.
095	3	Only scalar variables and array elements may appear in a set context.
096	3	The function xxx cannot appear in a set context.
097	3	xxx is followed by a parenthesized list but is not dimensioned and cannot be a function.
098	3	xxx is dimensioned and must appear in this context with subscripts.
099	3	xxx is an entry value and cannot appear in this context.

100	3	xxx cannot appear in this context.
101	3	Syntax error. A binary operator is required in place of xxx.
102	3	Syntax error. Unexpected occurrence of xxx.
103	1	Implementation restriction: The line number xxx must be less than 16384.
104	2	Missing end statement. One will be supplied by the compiler.
105	2	Syntax error. Statement consisting of only a statement label.
106	3	The character xxx is not a member of the FORTRAN character set or is out of place.
107	3	Invalid use of xxx.
108	2	Character string constant whose length is zero.
109	2	A character string constant has been terminated by the end of the statement.
110	3	Implementation restriction: More than xxx constants in this statement.
111	3	Text of this statement exceeds xxx characters.
112	3	A continuation line was encountered that was not preceded by an initial line.
113	3	There is no line number on this line: xxx
114	3	The rightmost six digits of xxx will be used as the line number.
115	3	The line number xxx is not greater than xxx.
116	3	Syntax error. Decimal point missing from end of an operator or logical constant.
117	3	.xxx. is not an operator or constant known to this compiler.
118	3	Missing or incomplete exponent field. The value zero will be used.
119	3	Integer constant xxx cannot be represented internally.
120	3	More than xxx digits in the floating point constant xxx.

121	3	Exponent overflow while converting the floating point constant xxx.
122	3	Exponent underflow while converting the floating point constant xxx.
123	3	Implementation restriction: More than xxx tokens in this statement.
124	3	Implementation restriction: Tokens are limited to xxx characters.
125	2	The data type of xxx must be the same as that of xxx.
126	3	xxx cannot be assigned an initial value.
127	3	The name xxx cannot be used as an entry point name.
128	3	xxx is a member of blank common and cannot be assigned an initial value.
129	2	The letter range specified in this statement overlaps a previous range.
130	3	xxx is contained in an octal constant but is not an octal digit.
131	3	A statement label appears on a line without any other text.
132	1	This statement contains a line with more than 80 characters.
133	3	A nonnumeric character was encountered in a label field.
134	2	Text appears after the closing right parenthesis of a format specification.
135	3	The variable xxx must be the index variable of a containing implied do loop.
136	3	Implementation restriction: The xxx of xxx exceeds 262143.
137	3	A prefix minus cannot precede xxx.
138	3	Implementation restriction: There are more than xxx arguments in this reference to xxx.

139	2	A character constant used to initialize xxx is longer than xxx characters.
140	3	Variable xxx has already been defined and cannot appear in a xxx statement.
141	2	xxx must be a scalar integer reference.
142	3	The named constant xxx must not appear in this context.
143	3	The expression starting with xxx must be a scalar or subscripted variable.
144	1	Warning: the meaning of multiple exponentiation has been changed from a previous release.
145	3	This xxx statement cannot have an input/output list.
146	3	This xxx statement must have an input/output list.
147	2	Variable xxx was declared with *-length, but is not a parameter or external function--length has been set to xxx.
148	1	Line xxx was interpreted as a comment, but a legal non-space character follows the initial c.
149	3	Asterisks designating external units are not permitted in xxx statements.
150	1	Unknown keyword xxx found in %global statement.
151	1	Unknown keyword xxx found in %options statement.
152	3	The terminating semicolon is missing from a %options or %global statement.
153	3	The concatenation operator may only be used if the ansi77 control argument or option was specified.
154	3	The substring operation may only be used if the ansi77 control argument or option was specified.
155	3	Invalid substring of xxx. Substring may only be applied to simple variables or array elements.

156	3	xxx cannot contain both ansi77 character variables and other data types.
157	3	An equivalence group cannot contain both ansi77 character variables and other data types.
158	3	Star-extent character strings may only be used if the ansi77 control argument or option was specified.
159	3	This statement contains a substring reference to xxx, which is not of the character data type.
160	3	The %include statement does not contain a file name.
161	3	The include file name xxx is longer than 19 characters. Statement ignored.
162	3	Include file xxx not found. Statement ignored.
163	3	Implementation restriction: only 255 include files allowed per compilation. Include file xxx ignored.
164	3	Implementation restriction: include file xxx exceeds the nesting limit of 32. Statement ignored.
165	3	This reference to xxx would cause infinite recursion of include files. Statement ignored.
166	3	An assumed-size array is not permitted in an xxx statement.
167	3	The array xxx has an assumed-size declarator in other than the upper bound of the last dimension.
168	3	A lower dimension bound of xxx is greater than the corresponding upper bound.
169	3	A dimension bound of xxx is neither a constant expression nor a scalar integer variable.
170	4	Compiler error: An invalid data type has been encountered during evaluation of a parameter expression.
171	3	A non-constant operand xxx was found while evaluating xxx.

172	3	An operator whose operands are of invalid type was found while evaluating xxx .
173	3	An attempt was made to use an unimplemented operation while evaluating xxx.
174	3	The xxx condition was raised during evaluation of xxx.
175	3	An invalid operator was found during evaluation of xxx.
176	3	An operand of invalid data type was found during evaluation of xxx.
177	4	Compiler error: the parameter statement work area has overflowed.
178	3	The block if beginning at line xxx has not been terminated.
179	3	The keyword xxx is missing in a xxx statement.
180	3	There is no block if statement corresponding to this xxx statement.
181	3	This xxx statement has followed an else statement in the same block if.
182	3	The do loop ending at xxx must be ended before this xxx statement.
183	3	This statement ends a do loop, but the do loop ending at xxx must be ended first.
184	3	This statement ends a do loop, but the block if at line xxx must be ended first.
185	3	The label xxx is on a statement that must not be referenced.
186	2	xxx is not the name of a common block.
187	3	Invalid specification or combination of specifications in a xxx statement: xxx.
188	2	The compiler is unable to get status information on source or include file xxx. The object segment will be nonstandard.
189	2	Implementation restriction: There are more than xxx arguments in this parameter list.

191	2	The xxx xxx supercedes xxx.
192	2	The control argument xxx supercedes the %global option xxx.
193	2	An invalid %global has been found and will be ignored. All % global must be at the beginning of the program.
200	4	Compiler error: the converter has encountered an unexpected operator with the op_code xxx.
201	4	Compiler error: attempt to increment_polish beyond end of polish input stack.
202	4	Compiler error: the converter work segment has overflowed while adding an entry to the xxx list.
203	3	Compiler error: an sf dummy arg has been found that does not match a known invocation.
204	3	Implementation restriction: an operator cannot use more than xxx operands.
205	4	Compiler error: converter work stack pointer has become negative.
206	4	Compiler error: converter work stack pointer has exceeded its upper bound of xxx.
229	3	Constant type not implemented.
300	1	xxx must be a scalar integer variable.
301	3	xxx must be a scalar integer variable.
302	3	The data type of a file expression, xxx, must be integer.
303	3	An internal file must have the character data type.
304	3	The record number expression, xxx, must be integer.
305	2	The encode or decode string may not be of logical data type.
306	3	Error detected in the definition of the statement function xxx.

307	3	Insufficient number of arguments in a reference to the statement function xxx.
308	3	Too many arguments supplied in a reference to the statement function xxx.
309	3	The data type of xxx is not compatible with its use.
310	3	The data type of xxx is not compatible with its use.
311	3	xxx is complex and xxx is double precision in an expression.
312	3	xxx is double precision and xxx is complex in an expression.
313	3	The data type of xxx must be logical.
314	3	The arguments to the builtin function xxx must have the same data type.
315	3	xxx does not have an arithmetic data type.
316	3	The do-loop control variable xxx cannot be complex.
317	3	xxx in a do statement must be arithmetic.
318	3	The do-loop control variable xxx must be arithmetic.
319	3	Wrong number of arguments in a reference to the builtin function xxx.
320	3	The builtin function xxx has an argument, xxx, that is not arithmetic.
321	3	Error in the use of the builtin function xxx. xxx has an invalid data type.
322	3	Error in the use of the builtin function xxx. xxx and xxx are complex values.
323	3	The complex value xxx cannot be used in this comparison.
324	3	xxx in a logical if, block if, or else if statement must be a logical value.
325	3	xxx in an arithmetic if statement must be an arithmetic value.

326	3	xxx in a computed if statement must be an arithmetic value.
327	3	Insufficient number of labels in a computed goto statement.
328	3	The complex values xxx and xxx cannot be used in this comparison.
329	2	The format item xxx must be an integer array or a character variable.
330	4	Compiler error: obsolete macro xxx occurs in the text.
331	3	The margin setting xxx must have the integer data type.
332	3	The filename xxx must be a character string.
333	3	The filetype xxx must be a character string.
334	3	In the use of the statement function xxx, xxx does not have the correct data type.
335	3	The file to be chained to xxx must be a character string or an integer array.
336	3	The system to be chained to xxx must be a character string or an integer array.
337	3	The character variable xxx cannot be assigned to an arithmetic variable.
338	3	xxx cannot be assigned to xxx, because it is not a logical variable.
339	3	The logical value xxx cannot be an operand of a relational operator.
340	3	The arithmetic value xxx can only be compared to another arithmetic value or a hollerith constant.
341	3	xxx does not have the character or integer data type and cannot be compared with xxx.
342	3	Error in processing the label list in a computed goto statement.
343	3	The data type of the statement function xxx must be arithmetic or logical.

344	3	Cannot convert a double precision expression to complex in the statement function xxx.
345	3	Cannot convert a complex expression to double precision in the statement function xxx.
346	3	Cannot convert the defining expression to the data type of the statement function xxx.
347	3	The statement function xxx does not have the proper data type.
348	1	This reference to xxx contains a character argument that may require descriptors.
349	3	An error has been detected in the processing of an open field.
350	3	The iostat variable xxx must have the integer data type.
351	3	The argument xxx used in this field must have the character data type.
352	3	The argument xxx used in this field must have the integer data type.
353	3	The argument xxx used in this field must have the logical data type.
354	3	xxx is used in a character expression but does not have the character data type.
355	3	The character valued function parameter xxx may not be declared to have *-length.
356	3	An error has been detected in the processing of an inquire statement field.
357	4	Compiler error: Invalid field number xxx encountered in inquire statement.
358	2	The format item xxx cannot be a Very Large Array.
380	3	An undefined label has been found in this program.
381	4	Compiler error: the output from the optimizer overwrites the next statement.
382	4	Compiler error: xxx does not agree with xxx.

383	4	Compiler error: The optimizer has encountered an unexpected operator with the op_code xxx.
384	1	The code from LINE xxx to LINE xxx is unreachable or unnecessary. It will not be compiled.
385	4	Implementation restriction: optimization has terminated due to lack of available bits in the masks.
386	4	Compiler error: Inconsistency found between an operator and the inputs chain of one of its inputs.
387	1	This loop has been eliminated because, after optimization, it has no effect on the result of the program.
388	4	Implementation restriction: flow_unit table overflow. Simplify flow of control: use do statements for looping.
389	1	This loop has no exit.
390	4	Implementation restriction: optimizer has created too many new operators.
391	4	Compiler error: an expression unthreaded by strength reduction is input to other expressions.
400	2	xxx has been called with an inconsistent number of arguments.
401	2	xxx is inconsistent with the corresponding argument type used in xxx.
402	4	Compiler error: an invalid index has been used with a xxx macro.
403	4	Compiler error: a return macro without arguments has been used to return from a func.
404	4	Compiler error: a return macro with an argument has been used to return from a proc.
405	4	Compiler error: an exit macro has been used to return from a func.
406	4	Compiler error: an exit macro has been used in a proc not invoked by scan.

407	4	Compiler error: xxx overflows its maximum of xxx words.
408	4	Compiler error: an s_return has been used when no matching s_call exists.
409	4	Compiler error: an ind_jump macro was used when the eaq was not in an indicator substate.
410	4	Compiler error: an if_ind or unless_ind macro was used when the eaq was in an invalid state.
411	4	Compiler error: an add_to_address macro was used with non-rel_constant xxx.
412	4	Compiler error: an attempt was made to get the character length of the noncharacter operand xxx.
413	4	Compiler error: there was an attempt to execute a nonexecutable macro.
414	4	Implementation restriction: xxx has overflowed its limit of xxx words.
415	4	Compiler error: the reference count of xxx has become less than 0.
416	4	Compiler error: xxx has an invalid address field.
417	4	Compiler error: xxx needs a pointer register for addressing, but is neither a parameter nor in common.
418	4	Compiler error: no index or pointer registers are available for allocation.
419	4	Compiler error: the offset of xxx cannot be found in storage.
420	4	Compiler error: an attempt was made to update the eaq with xxx to the ind state.
421	4	Compiler error: an attempt was made to load xxx into an invalid eaq state.
422	3	The subscript xxx of xxx is out of range.
423	3	The number of subscripts of xxx does not equal the number of its bounds.

424	4	Compiler error: xxx not implemented.
425	4	Compiler error: operand stack in improper state at end of subprogram.
426	1	The length for common block xxx has been increased to xxx words.
427	4	Compiler error: the address of xxx has been lost.
428	3	The entry point xxx has been multiply declared.
429	2	Multics restriction: The common block name xxx contains a dollar sign and cannot be initialized.
430	3	Compiler error: the value of xxx should be in the index register, but it has been lost.
431	1	The subscript xxx of parameter xxx is out of range.
432	2	The common block xxx is initialized more than once. The first initialization is used.
433	4	Implementation restriction: the product of xxx and xxx cannot be stored in the stack.
434	1	Common block xxx is declared with more than one length.
435	4	Compiler error: xxx must be a temporary or an array ref.
436	4	Compiler error: obsolete macro xxx occurs in the text.
437	4	Compiler error: a var proc is invoked by a call macro.
438	4	Compiler error: proc arg count not equal to actual arg count. xxx xxx
439	4	Compiler error: eaq not loaded by load_for_test macro.
440	4	Compiler error: eaq not loaded correctly for xxx.
441	4	Compiler error: operand xxx in return macro is not a temporary node.

442	4	Compiler error: temporary already has an address.
443	1	Only the first xxx characters of xxx can be used.
444	4	Compiler error: This statement cannot have a machine state associated with it.
445	4	Compiler error: xxx has an invalid xxx field.
446	4	Compiler error: xxx has an uninitialized address field.
447	4	Compiler error: an increment cannot be added to the address of xxx.
448	4	Compiler error: Could not put operand into EAQ machine state.
449	4	Compiler error: Both A and Q found already locked by eaq_man.
450	4	Compiler error: An operand that should be in the eaq was not found by get_eaq_name.
451	1	Implementation restriction: Global xxx table overflows. Optimization may be degraded.
452	4	Compiler error: data type xxx undefined for a call to create_constant.
453	4	Compiler error: attempt to load global item in a reserved register.
454	4	Compiler error: attempt to set up xxx while it has a nonpositive reference count.
455	4	Compiler error: an operator appears in the polish that should only appear in the quadruples.
456	3	Compiler error: the reference count of xxx was left too high.
457	3	The xxx appears in a substring reference to xxx, but falls outside the range of a legal character index.
458	3	The xxx which appears as a subscript of xxx does not have a numerical data type.

459	3	The xxx which appears in a substring reference to xxx does not have a numerical data type.
460	3	A substring reference to xxx has a constant length that is less than 1.
461	3	The xxx intrinsic function may not be used as an external function.
462	4	Compiler error: No integer constant was present on the operand stack when the int_to_char1 macro was invoked.
463	4	Compiler error: No character constant was present on the operand stack when the char1_to_int macro was invoked.
464	4	Compiler error: The optimizing code generator encountered the xxx intrinsic function.
465	4	Compiler error: The output of an operator called from a scan is not a temporary.
466	4	Compiler error: An emit_eis macro with the equal_lengths flag was encountered in which the length of the second operand was nonzero.
467	4	Compiler error: base_man_store_temp cannot find a usable pointer register.
468	3	xxx requires argument descriptors and may not be passed the assumed size array xxx.
469	2	Implementation restriction: Cannot initialize /xxx/ the definition section would overflow its limit of xxx words.
500	2	Cannot get pointer to subsequent source segment.
501	2	Number of symbols exceeds xxx. Symbol table will be processed in sections.
502	2	Compiler Error: Unknown node xxx discovered in symbol table.
503	2	The symbol xxx cannot be chained to others of the same name because the symbol table is too large.
523	4	Compiler error: VLA xxx has been encountered which is neither auto, static, common, nor parameter.

524	3	Development compiler error: xxx
525	4	Compiler error: unknown error xxx.

RUNTIME I/O ERROR MESSAGES

In a real situation, the messages listed below would occur accompanied by varying amounts of supplementary information, which might include:

(1) the specific item that is in error

(2) a description of the processing attempted by the FORTRAN runtime I/O routines when the error occurred

(3) where in the program the error occurred, and

(4) where in the file the error occurred.

Error messages include brief descriptive comments.

fortran_io: Error in access field.

fortran_io: This open attribute cannot be supplied if the file is already connected.

fortran_io: This open attribute cannot be supplied if the file is already opened.

fortran_io: Error in the attach description field.

fortran_io: Invalid or unexpected character in external data field.

fortran_io: Error in the blank field.

fortran_io: This file cannot be backspaced or rewound.

fortran_io: This file cannot be read.

fortran_io: This file cannot be opened with the requested mode.

fortran_io: This file opening does not permit file truncation.

fortran_io: This file opening does not permit output operations.

fortran_io: Error in the close statement attributes.

fortran_io: External data field cannot be converted.

fortran_io: Double word binary files are limited to double precision data.

fortran_io: Error in the filename field.

fortran_io: FORTRAN I/O Error. Contact FORTRAN maintenance personnel.

fortran_io: Error in the form field.

fortran_io: Error in format specification.

fortran_io: Infinite loop in format. There is a list item but the format has no field descriptors.

fortran_io: Formatted files are limited to formatted records.

fortran_io: The file opening is not compatible with the existing file.

fortran_io: An attempt has been made to access a record beyond the end of an internal file.

fortran_io: Only prompt, defer, and carriage attributes are allowed for file 0.

fortran_io: Only the print or terminal file type can be specified for file 0.

fortran_io: This operation is not allowed for file 0.

fortran_io: Invalid value for the maximum record length.

fortran_io: The value of a scale factor must be between -8 and 8 inclusive.

fortran_io: Error in the I/O switch field.

fortran_io: Maximum record length exceeded.

fortran_io: Namelist input must begin with a header.

fortran_io: Error in the mode field.

fortran_io: File must be empty in order to set maximum record length.

fortran_io: Error in namelist I/O.

fortran_io: This file is not a blocked file.

fortran_io: This file opening does not permit direct access I/O.

fortran_io: File must be open before being used.

fortran_io: This file was not created, opened, and attached by FORTRAN I/O.

fortran_io: This file opening does not permit sequential access I/O.

fortran_io: These two open attributes are mutually exclusive.

fortran_io: The open attributes are incomplete.

fortran_io: Maximum format parenthesis level exceeded.

fortran_io: Attempt to read more data than the record contains.

fortran_io: Error in status field.

fortran_io: Syntax error in the external data field.

fortran_io: Unformatted files are limited to unformatted records.

fortran_io: The file type of the external file is not recognized.

fortran_io: The I/O switch was not opened by FORTRAN and it does not support the requested mode.

MULTICS FORTRAN USER'S GUIDE
ADDENDUM C

SUBJECT

Changes to the Manual

SPECIAL INSTRUCTIONS

This is the third addendum to CC70, Revision 1, dated December 1979. Refer to the Preface for "Significant Changes."

Insert the attached pages into the manual according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margins indicate technical additions; asterisks denote deletions.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum C.

SOFTWARE SUPPORTED

Multics Software Release 10.2

ORDER NUMBER

CC70-01C

December 1983

39098
11183
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through viii ix, blank	iii through viii ix, blank
1-6.1, blank 1-7, 1-8	1-7, 1-8 1-8.1, 1-8.2
1-13, 1-14	1-13, 1-14
1-15 through 1-20	1-15 through 1-20 1-20.1, blank
1-23, blank 1-23.1, 1-24	1-23, 1-23.1 1-23.2, 1-24
3-3 through 3-6	3-3, 3-4 3-5, 3-5.1 3-5.2, 3-6
3-9, 3-10	3-9, 3-10
4-1 through 4-4	4-1 through 4-6 4-7, blank
B-9, B-10	B-9, B-10 B-10.1, blank
D-7 through D-22 D-23, blank	D-7 through D-22 D-23, blank
i-1 through i-8	i-1 through i-8 i-9, blank

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1983 File No.: 1L13, 1U13

LEVEL 68

MULTICS FORTRAN USERS' GUIDE ADDENDUM B

SUBJECT

Changes to the Manual

SPECIAL INSTRUCTIONS

This is the second addendum to CC70, Revision 1, dated December 1979. Refer to the Preface for "Significant Changes."

Insert the attached pages into the manual according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margins indicate technical additions; asterisks denote deletions.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum B.

SOFTWARE SUPPORTED

Multics Software Release 10.1

ORDER NUMBER

CC70-01B

February 1983

36163
7.5C183
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through ix, blank	iii through ix, blank
1-3 through 1-6	1-3 through 1-6
	1-6.1, blank
1-13, 1-14	1-13, 1-14
	1-14.1, blank
3-3 through 3-6	3-3, 3-4
	3-5, blank
	3-5.1, 3-6
3-7 through 3-10	3-7, blank
	3-7.1, 3-8
	3-9, 3-10
	3-10.1, blank
4-1 through 4-4	4-1 through 4-4
5-5 through 5-6	5-5, 5-6
5-17, 5-18	5-17, 5-18
	5-18.1, blank
6-3, 6-4	6-3, 6-4
i-1 thorugh i-8	i-1 through i-8

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

© Honeywell Information Systems Inc., 1983
2/83

File No.: 1L13, 1U13
CC70-01B

SERIES 60 (LEVEL 68)

MULTICS FORTRAN USERS' GUIDE ADDENDUM A

SUBJECT

This is the first addendum to CC70-01 (dated December 1979).

Insert the attached pages into the manual according to the collating instructions on the back of this cover.

Section 6 is new. In other sections, change bars indicate new and changed information; asterisks denote deletions. These changes will be incorporated into the next revision of this manual.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 9.1

ORDER NUMBER

CC70-01A

December 1981

33573
5C182
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
title page, preface	title page, preface
iii through vii, blank	iii through ix, blank
1-13 through 1-24	1-13 through 1-23, blank 1-23.1, 1-24
2-1, 2-2	2-1, 2-2
3-3 through 3-6	3-3 through 3-6.2
4-3, blank	4-3, blank
5-5 through 5-8	5-5 through 5-8.1, blank
	6-1 through 6-4
B-7 through B-12	B-7 through B-12
D-1 through D-14	D-1 through D-23, blank
i-1 through i-8	i-1 through 1-8

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

(c) Honeywell Information Systems Inc., 1981

File No.: 1L13

12/81

CC70-01A

PREFACE

The purpose of this manual is to supplement Multics FORTRAN, Order No. AT58.

Anyone faced with the prospect of learning to use an unfamiliar computer system is likely to experience some frustration in trying to get information out of the manuals that are supposed to explain it all. The inexperienced or occasional user is often at a loss for where to start understanding it all, especially since the manual explaining it all seems to assume everything. You want to know where there is a manual explaining how to use the manual that is supposed to explain it all.

The FORTRAN Users' Guide is written in the hope that all of you who want to write FORTRAN programs on the Multics system can get answers to basic questions both about the system and about the the FORTRAN dialect embodied on it.

If you are new to the system, whatever your level of sophistication as a programmer, the first section, "Introduction to Multics," provides a general overview of the system from the standpoint of FORTRAN programming. You are strongly encouraged to read through this section carefully before reading any other part of the manual.

Sections are so designed as to make them independent of each other. Depending on what you want to know, you can read the rest of the manual in any order you choose.

INDEX

! 2-3

$ 1-10, 2-3

; 2-5

2-5

A

Addressing
  and linking 1-8.2

Ampersand character 2-5

ansi66
  selection 3-5, 6-1

ansi77
  selection 3-5, 6-1

Arrays
  dimensions 4-3
  large arrays 1-8, 1-16, 1-19.2
  size 1-13, 4-3
  very large arrays 1-8, 1-13, 1-16, 1-19.2

Assembly listing 3-8
  see also Listing and Source listing

Asterisk character 2-3

B

Binder
  allocation of storage for variables 4-3
  analogous to linkage editor 1-31
  and -relocatable 3-7
  and block data subprograms 1-13
  automatic variables 1-31
  block data subprogram 1-13
  common 4-3
  common storage 1-13, 1-32
  effect on dynamic linking 1-31
  local 4-3
  search rules 1-31

Block data subprogram
  common variables 1-13

Block data subprograms 1-13

Bound segment 4-3

C

Calling sequence 1-15, B-2

Card-image format
  see Format

Command level 1-5, 2-1
  defined 1-5

Commands
  add_search_rules 1-11
  bind 1-31
  debug 2-7, 3-12
  fortran 3-1
  initiate 1-11
  list 1-14.1
  new_proc 1-16
  print_search_rules 1-11
  probe 2-7, 3-12
  release 1-23.1
  run 1-22, 1-23
  set_fortran_common 1-13, 1-14
    common block definitions 1-13
  set_search_rules 1-11
  start 1-23.1
  status 1-14.1
  terminate_refname 1-11

Comment character
  card-image format 2-8
  free-form format 2-3

Common blocks 1-12, 1-15, 3-9
  allocation of 1-13
  declaration of 1-14
  initialized by block data subprogram 1-13, 1-32
  initialized to zero 1-13
  initialized with set_fortran_common 1-13
  permanent 1-13, 1-14
  size 4-3
  storage of 1-8
  very large common 1-17

Common storage
  unlabeled 1-15

compatibility
  with non-FORTRAN programs C-1

Compiler 3-3
  control arguments
    see Control arguments
  error messages 3-2, 3-4.1
  invocation of 3-1
  listing 3-7
  metering 3-10
  optimizing 3-9
  options
    see Control arguments
  output of 3-1
  program units compiled separately 1-15, 1-21, 1-26, 1-19.1, 1-19.2
  program units compiled together 1-15, 1-19.2
  relocation 3-7
  required argument 3-1
  subscript checking 3-6
  symbol table 3-11

connection
  explicit 5-5, 5-18.1
  implicit 5-2
  of a unit 5-2

Constants
  Character-string
    card-image format 2-7
  Hollerith
    card-image format 2-7
  optimization 3-9

Continuation character
  card-image format 2-5, 2-8
  free-form format 2-5

Control arguments 3-4
  -brief_table 3-11
  -card 2-7, 3-10.1
  -check 3-3
  -check_multiply 4-7

Control arguments (cont)
-fold 3-10.1
-large_array 3-4
-line_numbers 2-7, 3-7
-list 3-7, 3-8, 3-11, 3-12
-long_profile 3-4
-map 3-7, 3-12
-no_check_multiply 4-7
-optimize 3-9, 3-12
-profile 3-10
-relocatable 3-7
-safe_optimize 3-6
-severityN 3-3
-subscriptrange 3-6
with -optimize 3-6
with -table or -brief table 3-6
-table 3-7, 3-8, 3-11, 3-12
-time_ot 3-9
-vla 3-4.1
-vla_parm 3-4.1
compiler options 3-1, 3-3, 3-6, 3-7, 3-8, 3-11
safe_optimize 3-9

Conversion
card-image to free-form format 2-7

Directory
as segment 1-2
segments listed in 1-2
working 3-2

Directory hierarchy
defined 1-2
file system 1-4
pathnames in 1-2, 1-5
search rules and 1-5
see also Search rules
segment located by entryname 1-5

Dollar-sign character 1-10, 2-3

Dynamic linking
ambiguous references 1-5, 1-11
common blocks 1-14
consequences of 1-11
defined 1-9
initiated segments 1-11
linker 1-8.2
reference names and 1-10
resolution of external references 1-8

D

Data types
complex 1-15
double precision 1-15

Debugging
-brief_table 3-11
-table 3-11
and -brief_table 3-11
and -subscriptrange 3-6
debug 3-11
full symbol table 3-11
interaction of -table and -optimize 3-12
line numbers 2-7, 4-2
probe 3-11

E

Efficiency B-10

End line 2-6

Entryname
defined 1-2

Entrypoint name
defined 1-10
main_ 1-12

Error messages
severity control 3-4.1

Exclamation mark 2-3

Executable code
  sharing of 1-8.2

External addresses
  patching unnecessary 1-8.2

External references
  see Dynamic linking

F

Fault, linkage 1-9, 1-14, 1-8.2

Faults
  linkage 1-31

File
  defined 1-2

Files 4-2

Format
  card-image 2-7, 3-10.1
    comments 2-8
    compiling 2-7
    continuation 2-8
    line numbers 2-8
  free-form 2-3, 2-5, 3-10.1
    comments 2-5
    continuation 2-5
    conversion to 2-7

FORTRAN
  binder 1-31
  compiler 2-6, 3-2, 3-6, 3-7, 3-8, 3-9, 3-10, 3-11
  source program
    card-image and free-form format 3-10.1
    card-image format 2-7
    compiling of 3-1
    correcting 3-2
    creating and editing of 2-1
    free-form format 2-3, 2-5
    input formats 2-3

FORTRAN (cont)
  source program
    listing 3-7
  source segment 3-3
    creating 2-1
    creating and editing of 2-1
    name of 3-1
  storage classes 1-15, 1-21, 1-19.1
  valid programs 1-18, 1-20, 1-19.2

FORTRAN source program
  free-form format 2-3

Free-form format
  see Format

H

Home directory
  as initial Working directory 1-3
  see Directory

I

I/O switch
  defined 5-22

implicit connection 5-2

Initializing variables
  to zero 1-19.1

Input data transfers 5-4

Input format
  control arguments 2-3

Input/Output
  as comments 5-1
  implicit 5-1
  input 5-4

Input/Output (cont)
  open statement 5-5, 5-8.1
  output 5-4

Inquire statement 5-18

Integer multiplications 4-6

L

Large Arrays
  see Arrays

Line number
  defined 2-7
  maximum 2-7

Linkage fault 1-9, 1-14, 1-31, 1-8.2

Linker 1-10, 1-14
  addressing 1-10
  entrynames
    search 1-10
  entrypoint names
    search 1-10
  reference name 1-22
  standard linkage mechanism 1-12

Listing 3-7
  -line_numbers 3-7
  -table 3-7
  assembly-like 3-8
  contents of 3-7
  control arguments
    -brief_table 3-11
    -list 3-7
    -map 3-7
    -table 3-7
  header 3-7
  see also Source listing, 3-7
  statement labels 3-7
  symbolic names 3-8

Listing segment 3-7

Load module
  vs Multics 1-8.1

M

Main program 1-12

O

Object segment 1-8.2
  contents of 1-8
  linkage section 1-8
  standard Multics 1-8

open statement
  binary stream files 5-15
  default terminal unit to file 5-16
  nonstandard units 5-16
  reversing defaults 5-17
  storage system files 5-10, 5-11, 5-12, 5-13
  tape connection 5-17
  terminal read 5-9
  terminal write 5-9

open statements
  terminal read/write 5-8.1

Optimization
  common subexpressions 3-9, B-4
  constant propagation 3-9
  dead assignments 3-9, B-10
  efficient coding B-10
  global B-4, B-10
  implied do-loops B-2
  invariant expressions 3-9, B-7
  invariant operations 3-9
  local B-1
    machine-dependent
      quick call B-2
    machine-independent B-1
  logical if statements B-1

Optimization (cont)
  operand region 3-10
  strength reduction 3-9, B-8

Optimizations
  global B-4

Optimizer 3-9
  evaluation of function
    references B-1

P

Pathname
  absolute 1-2
  relative 1-2
  see also Directory hierarchy

Pathnames
  entrynames as components of
    1-2

PL/I
  argument transmission C-1
  declaration for FORTRAN data
    types C-1

preconnection
  see implicit connection and
    explicit connection

Program units
  compiled separately 1-18
    main_ 1-12
  compiled together 1-12,
    1-18

Programs
  size of 4-2

Q

quit and start 1-23

R

Ready message 1-5

Records
  length and form 4-1

Reference name
  and entrypoint names 1-10
  defined 1-10
  dynamic linking and 1-10
  initiated 1-22
  state at end of run 1-23

Reference name table 1-8
  defined 1-10

Relocation
  of code, at run time 1-8.2
  see also Binder

Run unit 1-16
  and reference names 1-22,
    1-23
  as new environment 1-23
  defined 1-22
  discarded storage at end of
    run 1-23
  effect on automatic storage
    1-22
  effect on permanent common
    blocks 1-22
  free storage area 1-22
  initialization of free
    storage area 1-22

Run units
  resolution of external
    references 1-8
  see also Dynamic linking

S

Search rules
  defined 1-11
  manipulation of 1-11

Segment
  attributes 1-2
  defined 1-2
  executable object 3-1
  listing 3-7
  object 1-12, 3-2, 3-8
    addressing 1-10
    linkage section 1-8, 1-10
    merged by binder (bound)
        1-31
    one main per 1-12
  source 3-3
  stack 4-3
  standard Multics object 1-8

Semicolon 2-5

set_fortran_common command
    1-14, 1-23

Source listing 2-5, 3-7
  contents of 3-7
  control arguments
    -brief_table 3-11
    -list 3-7
    -map 3-7
    -table 3-7
  header 3-7
  see also listing
  statement labels 3-7.1
  symbolic names 3-8

Source program 3-2
  -line_numbers 3-7
  card-image 2-7
  creating and editing of 2-1
  free-form format 2-3, 2-5

Source segment
  creating and editing of 2-1,
      2-2

Special characters
  ampersand 2-5
  asterisk 2-3
  comment 2-3, 2-8
  comments 2-3
  continuation 2-5, 2-8
  dollar sign 1-10, 1-15, 2-3

Special characters (cont)
  exclamation mark 2-3
  semicolon
    in free-form format 2-5
  underscore 2-3

Stack Segment 4-3
  defined 1-7
  header 1-7
  illustrated 1-24
  size 4-3
  stack frames 1-7
    defined 1-7
    illustrated 1-24

Standard Multics object
    segment 3-1

Statement labels 2-5

Statements
  %global 1-20, 4-7
  %options 1-20, 4-7
  automatic 1-21
  common 1-15
  data 1-13, 1-21
  data transfer
    input 5-4
    output 5-4
  number allowed 4-2
  open
    examples of 5-8.1
    how to use 5-5
  pause 1-23.1
  save 1-20, 1-21, 1-22,
      1-19.2
    optimizer and 1-21
  size 4-2
  stop 1-23.1

Storage
  allocation of 1-20, 1-21,
      1-8.1, 1-19.2
    automatic 1-15
    common 1-15
    static 1-15
  automatic 1-15, 1-17, 1-26
    allocation of 1-15

Storage (cont)
automatic
in programs compiled separately 1-20
in programs compiled together 1-20
large and very large arrays 1-16, 1-19.2
quick call 1-17
classes 1-15
automatic 1-15
default
local 1-15
normal common 1-15
permanent common 1-15
static 1-15
common 1-13, 1-22
allocation of 1-16
state at end of run 1-23
common blocks 1-32
constraints on local static size 1-32
default 1-21
free storage area 1-7, 1-8
common blocks 1-8
linkage section 1-8, 1-9, 1-12
links 1-9
of external variables 1-13
reference name table 1-8
run unit 1-22
static variables 1-8, 1-16
initialization of 1-15, 1-8.1
local 1-20, 1-21, 1-22, 1-19.1
see automatic/static 1-15
static 1-20
local static
at end of run 1-23
managed storage 1-7, 1-8, 1-16, 1-17
normal common 1-15
permanent common 1-15
release of 1-20
Stack segment 1-7
static 1-8, 1-15

Storage (cont)
static
allocation of 1-16
common and local 1-17
large and very large arrays 1-16
very large common 1-17

Storage system
directory hierarchy 1-1
segment 1-2

Subscript errors 3-6

Symbolic names 2-3
common blocks 3-9
dollar sign entrypoint name 1-12
dollar sign in external 2-3
entrypoint names 3-8
external 3-8

T

terminal I/O 5-5

Text editor 2-1
qedx 2-2

U

Underscore character 2-3

unit
connection to file 5-1
default connection 5-3
defined 5-1
open statement 5-5

V

Variables
automatic 1-15, 1-20, 1-21

Variables (cont)
  automatic
    allocation of 1-19.1
    explicit initialization of
        1-20
    in programs compiled
        separately 1-20
    in programs compiled
        together 1-20
    initialization of 1-15
      programs compiled
          separately 1-18
      programs compiled
          together 1-18
    uninitialized 1-19.2
  external 1-12
    in free storage area 1-13
  initial values
    zero 1-20
  initial values of 1-13,
      1-21, 1-22, 1-19.2
    and data statement 1-21
    zero 1-19.2
  initial values of zero
      1-19.1
  initialized in data
      statement
    programs compiled together
        1-20
  local 1-20
    static 1-20
  static 1-15, 1-20, 1-22
    allocation of 1-20, 1-21
    initialization of 1-16
    storage of 1-8
  storage allocation of 1-8.1
  undefined 1-20, 1-19.1,
      1-19.2

Very Large Arrays
  see Arrays

Very Large Common
  see Common blocks

W

Working directory 3-2
  defined 1-3
  see Directory

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 68) MULTICS FORTRAN USERS' GUIDE	ORDER NO.	CC70-01
		DATED	DECEMBER 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE __________

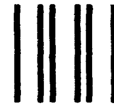
TITLE ______________________

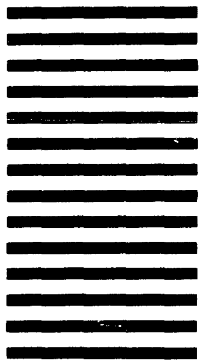
COMPANY ______________________

ADDRESS ______________________

____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms





BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

CUT ALONG LINE

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Kanda Jimbo-cho Chiyoda-ku, Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

26434, 3.5C983, Printed in U.S.A.

CC70-01