

# **FORTRAN for NOS/VE Topics for FORTRAN Programmers**

**GD**  
**CONTROL  
DATA**



**Usage**

60485916

# SCL Commands

---

## Managing Files:

<u>Operation</u>	<u>Command</u>
Create a file	CREATE_FILE (CREF)
Make a file available to a terminal session	ATTACH_FILE (ATTF)
Delete a local file	DETACH_FILE (DETF)
Delete a permanent file	DELETE_FILE (DELF)
Copy one file to another	COPY_FILE (COPF)
Create a catalog	CREATE_CATALOG (CREC)
Display a list of files in a catalog	DISPLAY_CATALOG (DISC)
Delete a catalog	DELETE_CATALOG (DELC)
Change the working catalog	SET_WORKING_CATALOG (SETWC)
Change the access permission of a file	CREATE_FILE_PERMIT (CREFP)

## Interactive Input and Output:

<u>Operation</u>	<u>Command</u>
Associate a file with the terminal	CREATE_FILE_CONNECTION (CREFC)

## Copying Files Between NOS and NOS/VE:

<u>Operation</u>	<u>Command</u>
Copy a file from NOS to NOS/VE	GET_FILE (GETF)
Copy a file from NOS/VE to NOS	REPLACE_FILE (REPF)

# **FORTRAN for NOS/VE**

## **Topics for FORTRAN Programmers**

### **Usage**

This manual describes a subset of the features and parameters documented in the following manuals:

**FORTRAN for NOS/VE Language Definition Usage Manual**  
**SCL for NOS/VE Language Definition Usage Manual**  
**SCL for NOS/VE System Interface Usage Manual**  
**SCL for NOS/VE Object Code Management Usage Manual**

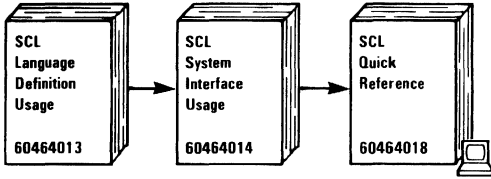
Control Data cannot be responsible for the proper functioning of any features or parameters not documented in these manuals.

**Publication number 60485916**

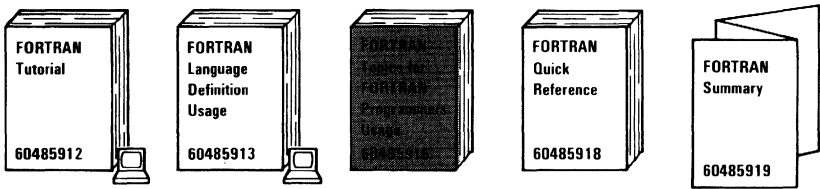
# Related Manuals

---

Background (Access as Needed):




FORTRAN Manual Set:



Additional References:



→ indicates the recommended reading sequence

 means available online

# Manual History

---

Revision A documents FORTRAN for NOS/VE Version 1 at release level 1.1.2, PSR level 630. It was printed in March, 1985.

©1985 by Control Data Corporation. All rights reserved.  
Printed in the United States of America.



# Contents

---

<b>About This Manual</b> .....	5
<b>Introduction to NOS/VE</b> .....	1-1
System Command Language .....	1-1
Files and Catalogs .....	1-2
Interactive Input and Output .....	1-20
Copying Files Between NOS and NOS/VE .....	1-24
Compiling and Executing a FORTRAN Program .....	1-25
Submitting Batch Jobs .....	1-29
Executing SCL Commands Inside a FORTRAN Program .....	1-33
Passing Parameters to a FORTRAN Program .....	1-34
Short Forms of SCL Commands .....	1-38
Summary of NOS/VE Capabilities .....	1-39
<b>Debugging</b> .....	2-1
The Debugging Process .....	2-1
Debugging Aids .....	2-20
<b>Introduction to FORTRAN Input/Output</b> .....	3-1
Basic Concepts .....	3-1
Overview of Input/Output Methods .....	3-4
Selecting an Input/Output Method .....	3-26
Summary of FORTRAN Input/Output Statements .....	3-31
<b>Optimizing a Program</b> .....	4-1
Basic Concepts .....	4-1
Optimization Techniques .....	4-3
FORTRAN Command Parameters .....	4-15
Summary of Optimizing a Program .....	4-19
<b>Using Virtual Memory</b> .....	5-1
What is Virtual Memory? .....	5-1
How Does Virtual Memory Work? .....	5-4
Programming Guidelines .....	5-8
Summary of Using Virtual Memory .....	5-17

<b>Using Object Libraries</b> .....	6-1
The Loading Process .....	6-1
What Are Object Libraries and Why Are They Useful?.....	6-7
How the Loader Uses Object Libraries .....	6-8
Creating and Modifying Object Libraries .....	6-14
Summary of Using Object Libraries .....	6-23
<b>Index</b> .....	Index-1

# About This Manual

---

This manual provides introductory information for users of the Control Data® FORTRAN Version 1 language under the Network Operating System/Virtual Environment (NOS/VE). This manual is a supplement to the FORTRAN for NOS/VE Language Definition manual. It emphasizes the use of FORTRAN within the NOS/VE environment.

## Audience

This manual is written for FORTRAN programmers who are new to NOS/VE but who have at least six months programming experience. The manual provides this audience with knowledge for developing, debugging, and executing programs under NOS/VE.

## Organization

The FORTRAN manual set consists of the following manuals: A tutorial, a language definition manual, a FORTRAN topics manual, and a summary. The tutorial is written for beginning FORTRAN programmers. It presents a simple introduction to basic FORTRAN topics. The language definition manual presents detailed descriptions and definitions of all FORTRAN statements and features. It is a complete reference to the NOS/VE FORTRAN language. The summary is a pocket-size booklet that presents a summary of the FORTRAN statements and features. The summary includes statement formats and parameter descriptions.

The Topics for FORTRAN programmers manual presents information intended to help FORTRAN programmers become familiar with NOS/VE and with the CDC version of FORTRAN. This manual is organized into the following chapters:

- Chapter 1, Introduction to NOS/VE, presents an introduction to NOS/VE commands and concepts. Topics include managing files, compiling and executing batch and interactive programs, and interactive input and output.
- Chapter 2, Debugging, presents an introduction to the debugging process. This chapter discusses common errors, what to do when those errors occur, and suggestions for avoiding errors. The chapter also describes debugging aids available to FORTRAN programmers, including the interactive debugging utility.

- Chapter 3, Introduction to FORTRAN Input/Output, summarizes and compares the various ways of performing input/output in a FORTRAN program. A discussion of the advantages and disadvantages of each method is included. This information is intended to help you decide when to use the various methods.
- Chapter 4, Optimizing a Program, describes how you can alter a program to make it execute faster. This chapter describes optimizations performed by the compiler and emphasizes techniques you can use to help the compiler perform those optimizations.
- Chapter 5, Using Virtual Memory, presents an introduction to the virtual memory concept used by NOS/VE. This chapter also includes guidelines for writing programs that use virtual memory more efficiently.
- Chapter 6, Using Object Libraries, describes how the system uses object libraries and how they can make a program more efficient. This chapter also explains how to create and modify object libraries.

## Conventions

Certain notations are used throughout this manual with consistent meaning. The notations are:

UPPERCASE	In language syntax, uppercase indicates a keyword, parameter, or symbol that must be written exactly as shown. (Although lowercase letters are interpreted the same as uppercase letters when used in FORTRAN keyword and symbols, uppercase is used in this manual for consistency. In occasional examples, keywords and symbols are shown in lowercase for illustrative purposes.)
lowercase	In language syntax, lowercase indicates a name, number, symbol, or entity that you must supply.
blue type	In examples of terminal dialog, blue type indicates user input.
...	In language syntax, a horizontal ellipsis indicates that the preceding optional item can be repeated as necessary.

⋮

In program examples, a vertical ellipsis indicates that other statements or parts of the program have not been shown because they are not relevant to the example.

Δ

Blank character. This symbol is used wherever there might otherwise be doubt as to how many blanks are intended.

|

In examples of formatted input and output, vertical bars denote the input or output fields. When used to enclose a numeric quantity, vertical bars indicate the magnitude (absolute value) of the quantity.

All numbers used in this manual are decimal unless otherwise indicated. Other number systems are indicated by a notation after the number. For example, 177 octal, FA34 hex.

## Ordering Manuals

Control Data manuals are available through Control Data sales offices or through:

Control Data Corporation  
Literature and Distribution Services  
308 North Dale Street  
St. Paul, Minnesota 55103

When ordering a manual, please specify the complete manual title and publication number. For example, FORTRAN for NOS/VE Language Definition Usage manual, publication number 60485913.

## Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation  
Publications and Graphics Division  
P. O. Box 3492  
Sunnyvale, CA 94088-3492

Please indicate whether you would like a written response.



A basic understanding of the NOS/VE System Command Language enables you to manage program and data file, compile and execute programs, and perform other useful functions in the NOS/VE environment.

## System Command Language

The System Command Language (SCL) is the language through which you communicate with the NOS/VE operating system.

SCL provides commands and statements that enable you to compile and execute your FORTRAN programs; to create, store, and retrieve the files containing your programs and the data used by those programs; and to perform a variety of other operations.

SCL is a language that is similar in many respects to FORTRAN. In addition to the SCL commands, it contains elements such as variables, constants, and operators, that can be combined into expressions and statements.

This chapter presents an overview of some of the more useful SCL commands and concepts. You should be familiar with these commands and concepts in order to effectively write and run programs under NOS/VE. For a complete description of all aspects of SCL, you should refer to the following manuals:

- SCL Language Definition Manual. This manual defines the complete SCL language specification. It assumes that you are unfamiliar with SCL, but are familiar with programming language concepts in general.
- SCL System Interface Manual. This manual describes the system interface to NOS/VE. It includes such topics as system access, interactive processing, file management, and resource management. This manual can be a useful resource for FORTRAN programmers running programs in the NOS/VE environment.
- SCL Quick Reference. This manual provides a quick reference for the commands described in the preceding two manuals. This manual describes SCL commands and parameters in alphabetical order. The SCL Quick Reference is also available as an online manual. You can enter this online manual by leaving this manual and typing

EXPLAIN M=SCL

These manuals provide a complete set of documentation for SCL. The rest of this chapter provides an overview of some of the more useful commands and concepts presented in those manuals.

## Files and Catalogs

Under NOS/VE, data is stored in units called files. Catalogs provide a useful way for you to organize your files. SCL provides commands that help you maintain and manage your files. Using these commands, you can create, store, delete, copy, and perform a variety of other operations on files.

### Files

A file is simply a collection of data that begins at a boundary known as the beginning-of-information (BOI) and ends at a boundary known as the end-of-information (EOI).

Many of the operations you perform during the course of creating and executing a FORTRAN program involve the manipulation of files. These files include the data files read and written by your program; they also include the file that contains your source program and the output files produced by the FORTRAN compiler.

NOS/VE recognizes two basic types of files: local (or temporary) files and permanent files. A local file exists only for the duration of a terminal session. When the session ends, the file is lost. In addition, any local files you create during a terminal session can be accessed only by you.

A permanent file exists after your terminal session ends, and continues to exist until you explicitly remove it from the system. You can retrieve a permanent file in subsequent terminal sessions. You can also authorize other users to access your permanent files.

A permanent file can have both a permanent name and a local name. A reference to the local name is equivalent to a reference to the permanent name.

### Catalogs

All files reside in catalogs. A catalog is simply a collection of files grouped under a catalog name. Catalogs serve two purposes: They provide a way for you to organize and keep track of your files, and they are used by NOS/VE to establish the type (local or permanent) of a file.



A basic understanding of the NOS/VE System Command Language enables you to manage program and data file, compile and execute programs, and perform other useful functions in the NOS/VE environment.

## System Command Language

The System Command Language (SCL) is the language through which you communicate with the NOS/VE operating system.

SCL provides commands and statements that enable you to compile and execute your FORTRAN programs; to create, store, and retrieve the files containing your programs and the data used by those programs; and to perform a variety of other operations.

SCL is a language that is similar in many respects to FORTRAN. In addition to the SCL commands, it contains elements such as variables, constants, and operators, that can be combined into expressions and statements.

This chapter presents an overview of some of the more useful SCL commands and concepts. You should be familiar with these commands and concepts in order to effectively write and run programs under NOS/VE. For a complete description of all aspects of SCL, you should refer to the following manuals:

- SCL Language Definition Manual. This manual defines the complete SCL language specification. It assumes that you are unfamiliar with SCL, but are familiar with programming language concepts in general.
- SCL System Interface Manual. This manual describes the system interface to NOS/VE. It includes such topics as system access, interactive processing, file management, and resource management. This manual can be a useful resource for FORTRAN programmers running programs in the NOS/VE environment.
- SCL Quick Reference. This manual provides a quick reference for the commands described in the preceding two manuals. This manual describes SCL commands and parameters in alphabetical order. The SCL Quick Reference is also available as an online manual. You can enter this online manual by leaving this manual and typing

EXPLAIN M=SCL

These manuals provide a complete set of documentation for SCL. The rest of this chapter provides an overview of some of the more useful commands and concepts presented in those manuals.

## Files and Catalogs

Under NOS/VE, data is stored in units called files. Catalogs provide a useful way for you to organize your files. SCL provides commands that help you maintain and manage your files. Using these commands, you can create, store, delete, copy, and perform a variety of other operations on files.

### Files

A file is simply a collection of data that begins at a boundary known as the beginning-of-information (BOI) and ends at a boundary known as the end-of-information (EOI).

Many of the operations you perform during the course of creating and executing a FORTRAN program involve the manipulation of files. These files include the data files read and written by your program; they also include the file that contains your source program and the output files produced by the FORTRAN compiler.

NOS/VE recognizes two basic types of files: local (or temporary) files and permanent files. A local file exists only for the duration of a terminal session. When the session ends, the file is lost. In addition, any local files you create during a terminal session can be accessed only by you.

A permanent file exists after your terminal session ends, and continues to exist until you explicitly remove it from the system. You can retrieve a permanent file in subsequent terminal sessions. You can also authorize other users to access your permanent files.

A permanent file can have both a permanent name and a local name. A reference to the local name is equivalent to a reference to the permanent name.

### Catalogs

All files reside in catalogs. A catalog is simply a collection of files grouped under a catalog name. Catalogs serve two purposes: They provide a way for you to organize and keep track of your files, and they are used by NOS/VE to establish the type (local or permanent) of a file.

Everyone who is validated to use the computer automatically has two catalogs: one for local files and one for permanent files. (Initially, of course, the local catalog contains no files. The permanent file catalog may contain some default files, such as PROLOG and EPILOG.)

The local catalog is named \$LOCAL. This catalog contains all your local files. Any files that you create during a terminal session are assigned to the local catalog unless you explicitly assign them to a permanent file catalog. A file must be in the \$LOCAL catalog before it can be referenced in a FORTRAN program. The command to place a permanent file in the \$LOCAL catalog is described later in this chapter.

The permanent file catalog is known as the master catalog. (The master catalog that you are automatically assigned is all that you will typically need, although you can make special arrangements with your site administrator to have more than one permanent file catalog.)

The name of your master catalog is the same as your user name. However, you can also reference this catalog by the name \$USER. (\$USER is actually an SCL function that returns your user name.)

Any files that you place in your master catalog become permanent files. The command to place a file in the master catalog is described later in this chapter.

## Copying Files

The first command we will introduce is the COPY\_FILE command. This command copies the contents of a file to another file. COPY\_FILE is a commonly used command, and is used in examples throughout this chapter. The general form of the COPY\_FILE command is

```
COPY_FILE FROM=file-1 TO=file-2
```

Files file-1 and file-2 can be either local files or permanent files. If the destination file file-2 does not already exist, a new file is automatically created.

For example, the command

```
COPY_FILE FROM=OLDFILE TO=NEWFILE
```

copies local file OLDFILE to local file NEWFILE.

## Referencing Files

All files under NOS/VE are identified by a name that consists of 1 through 31 letters, digits, and the characters \$, \_, @, and #. You reference a file simply by specifying its name and the catalog in which it resides. The general form of a file reference is

**catalog-name.file-name**

The catalog name and the file name are separated by a period.

For example, the following command copies the contents of a local file named OLDFILE to a local file named NEWFILE:

```
COPY_FILE FROM=$LOCAL.OLDFILE TO=$LOCAL.NEWFILE
```

You can also reference a permanent file in this way. For example, the file reference

```
$USER.BIRD
```

references a file named BIRD in the master catalog.

Specifying a catalog name and file name makes it possible for you to reference permanent files without the need for entering a system command to make the files available to the terminal session.

The following example shows a file reference used in a COPY\_FILE command:

```
COPY_FILE FROM=$LOCAL.OLDFILE TO=$USER.NEWFILE
```

This command copies the contents of the local file OLDFILE into the permanent file NEWFILE, which resides in the master catalog. (If NEWFILE does not already exist, it is created.)

If you specify only a file name, without prefixing that name with a catalog name, the file is assumed to be in a catalog known as the working catalog.

More on the working catalog later. For now, we'll assume that the working catalog is the \$LOCAL catalog. (The system assumes this unless you explicitly request otherwise.) For example, simply specifying the file name BIRD is the same as specifying

```
$LOCAL.BIRD
```

In the following example, local file OLDFILE is copied to permanent file NEWFILE:

```
COPY_FILE FROM=OLDFILE TO=$USER.NEWFILE
```

An important restriction on this method of file referencing is that it can be used only in system commands. In FORTRAN input/output statements, you can specify only the file name; you cannot prefix that name with a catalog name. For example, in an OPEN statement, you can specify

```
OPEN (UNIT=2, FILE='BIRD')
```

but not

```
OPEN (UNIT=2, FILE='$USER.BIRD') <---- WRONG!
```

Since you can't specify catalog names in a program, it follows that files referenced in a program must be local files (in the \$LOCAL catalog). The commands for making permanent files available to a program as local files are discussed later in this chapter.

## Creating Files

You can create files during a terminal session in two ways. The first way is simply to reference a nonexistent file. The system will automatically create the file. This is known as creating a file implicitly. For example, in the command

```
COPY_FILE FROM=OLD_FILE TO=NEW_FILE
```

the system creates NEW\_FILE (if it does not already exist) and copies the contents of OLD\_FILE to NEW\_FILE. In this case, NEW\_FILE is a local file.

You can implicitly create permanent files by specifying the catalog name in the file reference. For example, the command

```
COPY_FILE FROM=OLD_FILE TO=$USER.NEW_FILE
```

creates a file named NEW\_FILE in the master catalog.

You can also implicitly create files in a FORTRAN program. If you specify a nonexistent file in an input/output statement, that file is automatically created as a local file. Note, however, you cannot create permanent files in this way. The reason is that FORTRAN does not allow you to specify catalog names along with the file name. For example, the sequence

```
OPEN (UNIT=2, FILE='FORT_FILE')
WRITE (UNIT=2) X, Y, Z
```

implicitly creates a local file named FORT\_FILE (assuming it does not already exist) and writes the values of variables X, Y, and Z to that file.

If you want to save a local file for use in a subsequent terminal session, you must make a permanent copy of the file. To do this, you can use the following method.

The second way of creating a permanent file is to use the `CREATE_FILE` command. This command creates an empty permanent file. You can then save a local file by copying the contents of that file to the permanent file.

The simplest form of the `CREATE_FILE` command is

```
CREATE_FILE FILE=file-reference
```

where `file-reference` specifies the name of the file and the catalog in which the file is to reside. For example, the command

```
CREATE_FILE FILE=$USER.NEW_FILE
```

creates an empty permanent file named `NEW_FILE` in the master catalog.

The next step in saving a local file is to copy the contents of the local file to the permanent file. To do this, you can use the `COPY_FILE` command to copy the contents of the local file into the empty permanent file. The `COPY_FILE` command has the form

```
COPY_FILE FROM=file1 TO=file2
```

where `file1` is the file to be copied and `file2` is the file to receive the data.

For example, the command to copy the data from the local file `LOC_FILE` to the permanent file `NEW_FILE` created by the `CREATE_FILE` command is

```
COPY_FILE FROM=LOC_FILE TO=$USER.NEW_FILE
```

## **Making Permanent Files Available to a Program**

To reference a permanent file in a system command, you can simply prefix the file name with the catalog name. For example, the command

```
COPY_FILE TO=$USER.NEWFILE FROM=$USER.OLDFILE
```

copies file `OLDFILE` to file `NEWFILE`. Both permanent files are in the master catalog.

However, you cannot use this method of file referencing in a FORTRAN program. All files referenced in a FORTRAN program must be in the \$LOCAL catalog. There are two ways in which you can make a permanent file available in the \$LOCAL catalog. The first way is to attach the file by using an ATTACH\_FILE command. This command has the form

```
ATTACH_FILE FILE=file
```

where file specifies the name and catalog of the file to be attached. The ATTACH\_FILE command makes the specified file available in the \$LOCAL catalog and, therefore, available to a FORTRAN program.

For example, the command

```
ATTACH_FILE FILE=$USER.NEWFILE
```

makes permanent file NEWFILE available in the \$LOCAL catalog.

It is important to note that the ATTACH\_FILE command makes a permanent file available in the \$LOCAL catalog, but does not create a new copy of the file. There is still only one copy of the file. Thus, any changes you make to the file in the \$LOCAL catalog are also made to the file in the master catalog, because they both refer to the same file. Therefore, you do not need to take any further action to ensure that the changes you make to the local file are permanent.

If you want to write to a file, you must attach the file with WRITE permission. The ACCESS\_MODE parameter on the ATTACH\_FILE command specifies the permissions with a permanent file is attached. For example, following command makes file AFILE available through the \$LOCAL catalog:

```
ATTACH_FILE FILE=$USER.AFILE ACCESS_MODE=(READ,EXECUTE,WRITE)
```

File AFILE is now available in both the \$LOCAL and master catalogs. The file is attached with read, execute, and write permissions. The following two commands are equivalent:

```
COPY_FILE FROM=$USER.AFILE TO=BFILE <-Accesses AFILE through  
master catalog
```

```
COPY_FILE FROM=AFILE TO=BFILE <-----Accesses AFILE through the  
$LOCAL catalog
```

The following command accesses AFILE through the \$LOCAL catalog, but changes the only existing copy of AFILE. Thus, the changes to AFILE are permanent.

```
COPY_FILE FROM=CFILE TO=AFILE
```

## PERMANENT FILES

The following example shows how a permanent file in the master catalog is made available to a FORTRAN program:

```
PROGRAM READFL
OPEN (UNIT=1, FILE='NEWFIL') <-----Program looks for a local file
READ (UNIT=1, FMT=100) VALS          named NEWFIL.
```

The following command, entered before the program is executed, makes NEWFIL available in the \$LOCAL catalog:

```
ATTACH_FILE FILE=$USER.NEWFIL
```

NEWFIL is now available to the FORTRAN program.

The preceding paragraphs described how to use the ATTACH\_FILE command to make permanent files available to FORTRAN programs. The alternate way of making permanent files available is to use the COPY\_FILE command to create a temporary copy of a permanent file in the \$LOCAL catalog.

For example, assume you have a permanent file named WHICH\_FILE in your master catalog. You can create a temporary copy of WHICH\_FILE by entering the command

```
COPY TO=WHICH_FILE FROM=$USER.WHICH_FILE
```

This command creates a new and separate copy of WHICH\_FILE in the \$LOCAL catalog. Note that there are now two separate copies of WHICH\_FILE. Any changes you make to the local copy do not affect the permanent copy. To make those changes permanent, you must copy the temporary file to a permanent file. For example, the command

```
COPY_FILE TO=$USER.WHICH_FILE FROM=WHICH_FILE
```

copies the local copy of WHICH\_FILE over the permanent copy of WHICH\_FILE.

The following discussion may clear up some confusion about local and permanent files.

A given file can exist in more than one catalog. The file can have the same name in each catalog, or it can have different names. But regardless of whether the names are the same or different, they all refer to the same file. For example, the following commands make an existing permanent file available through the \$LOCAL catalog under two names:

```
ATTACH_FILE FILE=THIS_FILE
ATTACH_FILE FILE=THIS_FILE LOCAL_FILE_NAME=NEW_NAME
```



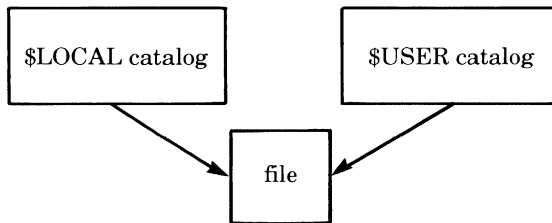
Permanent file `THIS_FILE` can now be referenced by either of the local file names `THIS_FILE` or `NEW_NAME`.

Different files in different catalogs can have the same name. For example, the commands

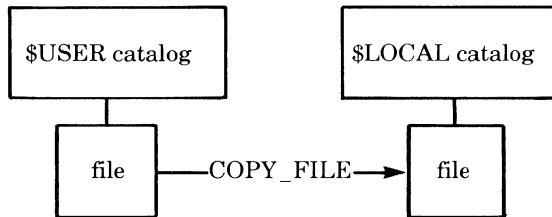
```
CREATE_FILE FILE=$USER.NEWFILE
COPY_FILE FROM=OLDFILE TO=NEWFILE
```

create a new (empty) file in the master catalog and copy an existing file to a new file in the `$LOCAL` catalog. Now, two different files exist, both named `NEWFILE`.

The following diagrams illustrate the `ATTACH_FILE` and `COPY_FILE` commands:



The `ATTACH_FILE` command makes the same copy of a permanent file available through the `$LOCAL` catalog.



The `COPY_FILE` command can create a local copy of a permanent file.

## Deleting Files

After you are through with a file, you can delete it from the system. Once you have deleted a file, that file can no longer be accessed and it cannot be recovered. The storage space occupied by the file becomes available for other uses. There are two commands for deleting files: one for local files and one for permanent files.

## DELETING FILES

The command to delete a local file is

```
DETACH_FILE FILE=local-file
```

This command removes the specified local file from the system. If the specified file is an attached permanent file, the permanent file is not affected.

For example,

```
DETACH_FILE FILE=$LOCAL.RFILE
```

removes (detaches) local file RFILE from the system.

The command to delete a permanent file is

```
DELETE_FILE FILE=perm-file
```

For example, the command

```
DELETE_FILE FILE=$USER.OFL
```

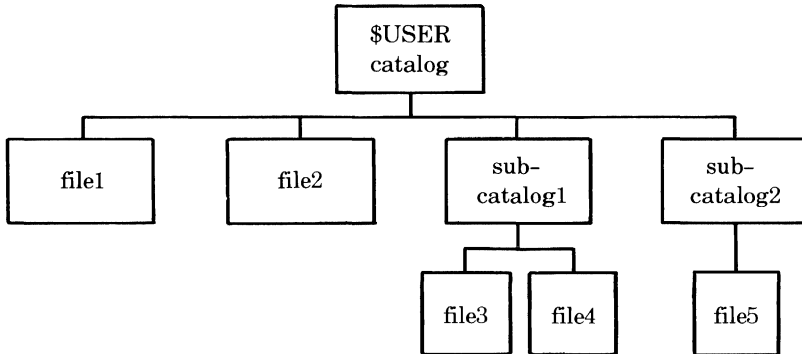
deletes permanent file OFL.

Deleting an attached permanent file (that was specified in an ATTACH\_FILE command) also deletes it from the \$LOCAL catalog, because there is really only one copy of the file. Note, however, that if you have created a separate copy of the permanent file in the \$LOCAL catalog, the local copy continues to exist after the DELETE\_FILE command.

## Creating Subcatalogs

You can create subcatalogs within your master catalog. A subcatalog is simply a catalog within a catalog. Subcatalogs provide a way of organizing your files into meaningful groups. You can't create subcatalogs in the \$LOCAL catalog.

The following diagram shows a master catalog that contains two files and two subcatalogs. The first subcatalog contains two files, and the second subcatalog contains one file.



The command to create a subcatalog has the form

**CREATE\_CATALOG CATALOG=master-catalog.subcatalog**

where master-catalog is the existing master catalog and subcatalog is the new subcatalog to be created. You can create a subcatalog within a subcatalog by specifying

**CREATE\_CATALOG  
CATALOG=master-catalog.subcatalog-1.subcatalog-2**

Each time you create a new subcatalog, you must specify the complete sequence of catalogs leading up to the new subcatalog.

For example, the following command creates a subcatalog named ANIMALS in the master catalog:

**CREATE\_CATALOG CATALOG=\$USER.ANIMALS**

The following commands create two subcatalogs, named FISH and BIRDS, in subcatalog ANIMALS:

**CREATE\_CATALOG CATALOG=\$USER.ANIMALS.FISH**

**CREATE\_CATALOG CATALOG=\$USER.ANIMALS.BIRDS**

We now have a master catalog that contains two levels of catalogs. The following commands create some files in subcatalogs FISH and BIRDS.

```
CREATE_FILE FILE=$USER.ANIMALS.FISH.CARP  
CREATE_FILE FILE=$USER.ANIMALS.FISH.GUPPY  
CREATE_FILE FILE=$USER.ANIMALS.BIRDS.ROBIN  
CREATE_FILE FILE=$USER.ANIMALS.BIRDS.PARROT
```

Subcatalog FISH contains files CARP and GUPPY, and subcatalog BIRDS contains files ROBIN and PARROT.

To reference a permanent file, you must specify the entire chain of catalogs leading to that file. For example, the following LIST\_FILE command references file PARROT:

```
LIST_FILE $USER.ANIMALS.BIRDS.PARROT
```

Note that the following reference to file PARROT is not valid, because it does not specify the entire chain of catalogs leading to PARROT:

```
LIST_FILE $USER.PARROT
```

## Displaying a List of Files in a Catalog

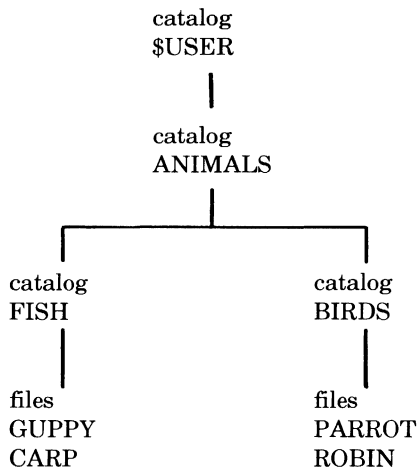
You can list the names of the files belong to a catalog by using the DISPLAY\_CATALOG command. This command has the form

```
DISPLAY_CATALOG CATALOG=catalog-name
```

The name you specify must be in the form \$USER.subcatalog-1.subcatalog-2. . . where subcatalog-n is the lowest level of subcatalog you want to list.

If you enter DISPLAY\_CATALOG without any parameters, the files in the \$LOCAL catalog are listed. (Any files created by a FORTRAN program which have not been referenced by an SCL command will not appear in the list.)

Our next example assumes the existence of the following catalogs:



Some DISPLAY\_CATALOG commands for this catalog are as follows:

<u>Command</u>	<u>Displayed at the Terminal</u>
DISPLAY_CATALOG CATALOG=\$USER	CATALOG :ANIMALS
DISPLAY_CATALOG CATALOG=\$USER.ANIMALS	CATALOG :FISH CATALOG :BIRDS
DISPLAY_CATALOG CATALOG=\$USER.ANIMALS.FISH	CATALOG :FISH FILE :CARP FILE :GUPPY
DISPLAY_CATALOG CATALOG=\$USER.ANIMALS.BIRDS	CATALOG :BIRDS FILE :ROBIN FILE :PARROT

## Deleting Catalogs

If you no longer need a particular permanent file catalog, you can delete the catalog by entering the command

```
DELETE_CATALOG CATALOG=$USER.subcatalog-1.
subcatalog-2. . . .
```

You can delete only an empty catalog (that is, a catalog that contains no files or subcatalogs). If you try to delete a catalog that contains files or subcatalogs, the `DELETE_CATALOG` command returns an error message and takes no action. For example, assuming the existence of the catalogs listed in the preceding example, the command

```
DELETE_CATALOG CATALOG=$USER.ANIMALS
```

will not affect the `ANIMALS` catalog as long as it contains files `ROBIN` and `PARROT`. However, if you first enter the commands

```
DELETE_FILE FILE=ROBIN
DELETE_FILE FILE=PARROT
```

and then enter

```
DELETE_CATALOG CATALOG=$USER.ANIMALS.BIRDS
```

the `BIRDS` catalog will be deleted.

You cannot delete the `$USER` and `$LOCAL` catalogs.

## Changing the Working Catalog

As you may have noticed, referencing files under `NOS/VE` can be unwieldy, especially if the files reside within several levels of catalogs. For instance, in our preceding example, the command to copy the contents of file `GUPPY` to file `PARROT` is

```
COPY_FILE TO=$USER.ANIMALS.BIRDS.PARROT FROM=$USER.ANIMALS.
FISH.GUPPY
```

To avoid specifying a chain of catalogs, you could use `COPY_FILE` to create local copies of the files, or use `ATTACH_FILE` to make the files available in the `$LOCAL` catalog. It would then be unnecessary to specify catalog names. (Recall that if you omit the catalog name from a file reference, the system searches for the file in the `$LOCAL` catalog.) However, a quicker way of referencing permanent files is available through the `SET_WORKING_CATALOG` command.

The working catalog is the catalog the system searches whenever you specify a file name with no catalog names. The default working catalog is the \$LOCAL catalog. For example, in the command

```
DISPLAY_FILE FILE=THIS_FILE
```

the system looks for THIS\_FILE in the \$LOCAL catalog.

You can change the working catalog to one of your permanent file catalogs by specifying a SET\_WORKING\_CATALOG command. Then, whenever you specify a file name with no catalog names, the system searches for the file in the permanent file catalog. This capability provides a shorthand way of referencing permanent files.

The SET\_WORKING\_CATALOG command has the form

```
SET_WORKING_CATALOG CATALOG=catalog-name
```

For example, assume the catalog \$USER.ANIMALS.BIRDS contains a file named PARROT. The command

```
SET_WORKING_CATALOG CATALOG=$USER.ANIMALS.BIRDS
```

sets the working catalog to \$USER.ANIMALS.BIRDS. Now, whenever you specify a file name with no catalog names, the system automatically appends the string \$USER.ANIMALS.BIRDS to the file name. Thus, specifying the file name PARROT is equivalent to specifying \$USER.ANIMALS.BIRDS.PARROT.

Note that after you have changed the working catalog to one of your permanent file catalogs, you must prefix all references to local files by \$LOCAL.

An important limitation on the SET\_WORKING\_CATALOG command is that it applies only to file names specified in SCL commands. Files referenced in FORTRAN programs MUST be in the \$LOCAL catalog, regardless of the current working catalog.

## Positioning Files

Whenever you reference a file in a system command, that file will, in most cases, be automatically positioned at the beginning-of-information before the command is executed. (You can change this default file positioning as explained later in this chapter.) However, you can override the default positioning for a particular reference to a file by referencing the file in the following way:

file.position

where file is a file name (which can be prefixed by catalog names) and position is one of the following:

**\$BOI**     The file is positioned at beginning-of-information.

**\$ASIS**    The file is left at its current position.

**\$EOI**     The file is positioned at end-of-information.

For example, in the following FORTRAN command, file PROG\_FILE is positioned at the beginning-of-information, file ERR positioned at end-of-information, and file LGO is left at its current position. The positioning occurs before the FORTRAN command is executed.

```
FORTRAN I=PROG_FILE E=ERR.$EOI B=LGO.$ASIS
```

Files referenced in a FORTRAN program during execution are not repositioned before the reference. (They remain at \$ASIS.) Therefore, if you want to read or write a file starting at the beginning-of-information, you should specify a REWIND\_FILE command before executing the program, or include a REWIND statement in the program before the READ or WRITE statement.

The file you specify to execute the program (for example, LGO) is positioned according to the system default (usually \$BOI).



## Permanent File Cycles

Each permanent file can have up to 999 different versions, or cycles. Each cycle is indicated by an integer that you append to the file name. (If you also specify a file position indicator, the cycle number goes between the file name and the position indicator.) For example, the specification

```
AFILE.1
```

references cycle 1 of file AFILE. The specification

```
$USER.INDAT.12.$BOI
```

references cycle 12 of file INDAT.

If you omit the cycle number, the system assumes the highest cycle that you have defined for the file.

Permanent file cycles provide a convenient way of storing and referencing different versions of a file. For example, suppose you are developing a program and you want to keep permanent copies of successive versions of the program. Assuming the program is on local file TEMP, you could create a permanent file named PGRAM as follows:

```
CREATE_FILE $USER.PGRAM <---- Create permanent file PGRAM
                               (initially empty).
COPY_FILE TEMP $USER.PGRAM <- Copy contents of local file TEMP into
                               permanent file PGRAM.
```

You now have cycle 1 of permanent file PGRAM. The following commands create cycle 2 of file PGRAM and copy information into it (assuming local file TEMP now contains a new version of the program):

```
CREATE_FILE FILE=$USER.PGRAM
COPY_FILE FROM=TEMP TO=$USER.PGRAM
```

Omitting the cycle number from the file reference in the CREATE\_FILE command causes the system to create the next cycle, in this case cycle 2. Omitting the cycle number from the file reference in the COPY\_FILE command causes the system to use the highest cycle defined for the file.

Now, you have two cycles of file PGRAM. The following commands are equivalent; they both attach the most current cycle (cycle 2):

```
ATTACH_FILE $USER.PGRAM.2
```

```
ATTACH_FILE $USER.PGRAM
```

To reference the original version (cycle 1), you must specify the cycle number:

```
ATTACH_FILE $USER.PGRAM.1
```

## Accessing Other Users' Files

Normally, your permanent files can be accessed only by you. However, you can make your files available to other users by using the `CREATE_FILE_PERMIT` command to change the access permission of the file. Two useful forms of this command are:

```
CREATE_FILE_PERMIT FILE=file-name USER=user-name
```

This command makes the specified file available to the specified user.

```
CREATE_FILE_PERMIT FILE=file-name GROUP=PUBLIC
```

This command makes the specified file public; that is, it is available to all users.

For example, the command

```
CREATE_FILE_PERMIT FILE=$USER.PGRAM USER=URNAME
```

makes permanent file PGRAM available to the user whose user name is URNAME.

By default, the designated user or users may read or execute the file, but may not alter it. Other parameters on the `CREATE_FILE_PERMIT` command allow you to change the access mode to permit the file to be altered.

After you have been given access permission to a file, you can reference that file by prefixing the file name by the user name of the user who owns the file. For example, assume your user name is MYNAME and you have entered the preceding `CREATE_FILE_PERMIT` command. User URNAME can now reference file PGRAM by specifying

```
.MYNAME.PGRAM
```

The period preceding MYNAME is required.

If a file is in a subcatalog within the master catalog, a user who accesses the file must specify the chain of subcatalogs leading to the file. For example, assume that file AAA is in subcatalog SUBCAT of your master catalog and that your user name is MYNAME. A user who has access permission to the file can reference that file by specifying

```
.MYNAME.SUBCAT.AAA
```

For example, if user URNAME enters the command

```
ATTACH_FILE FILE=.MYNAME.SUBCAT.AAA
```

that user will have access to file AAA through his or her \$LOCAL catalog.

## File Attributes

Each NOS/VE file has an associated set of attributes. These attributes define the structure of the file and establish various processing options for the file. Each attribute is identified by a unique name that you can use to reference the attribute in system commands. There are about 40 attributes altogether (although not all attributes apply to all files). Some examples of file attributes are:

FILE_ORGANIZATION	Specifies the organization of the file (sequential, indexed sequential, or byte addressable).
RECORD_TYPE	Specifies the record structure of the file.
ACCESS_MODE	Specifies how the file can be used by subsequent commands. For example, whether the file can be read, executed, or modified.
OPEN_POSITION	Specifies where the file is to be positioned before it is opened.

The attributes of a file are initially established when the file is created. Some of the attributes can later be changed, while others are permanent for the life of the file.

For most types of FORTRAN input/output, you do not need to be concerned with file attributes. The reason is that the files you'll be reading and writing are assigned default attributes that are appropriate for the file. The exception is the FORTRAN file interface capability, a highly flexible type of input/output that requires you to know many of the file attributes. For other types of input/output, you will generally use the default attributes. You can, however, change certain attributes.

The following commands are useful for working with file attributes:

<code>SET_FILE_ATTRIBUTES</code>	Sets one or more attributes for a file.
<code>CHANGE_FILE_ATTRIBUTES</code>	Changes one or more attributes for a file.
<code>DISPLAY_FILE_ATTRIBUTES</code>	Displays one or more attributes of a file.

More information about file attributes and the preceding commands is available in the FORTRAN Language Definition manual and the SCL System Interface manual.

## Interactive Input and Output

Under NOS/VE, you can write a program that performs interactive input and output; that is, one that reads data input from the terminal and displays output at the terminal.

You can perform interactive input and output in two ways. The first way is to use the standard system files `$INPUT` and `$OUTPUT`. These files are automatically associated with the terminal; that is, all data written to file `$OUTPUT` is displayed at the terminal, and all data read from file `$INPUT` is read from the terminal.

The second way is to use the `REQUEST_TERMINAL` command to associate a file name with the terminal. Then, any data written to that file is displayed at the terminal, and any data read from that file must be entered through the terminal.

Files `$INPUT` and `$OUTPUT` are standard system files that are available to all users. You do not need to enter any special commands to make these files available to your program. You can reference `$INPUT` and `$OUTPUT` in system commands as well as within a FORTRAN program. For example, assuming file `DISFILE` contains printable data, the following command displays the contents of `DISFILE` at the terminal:

```
COPY_FILE FROM=DISFILE TO=$OUTPUT
```

Referencing files `$INPUT` and `$OUTPUT` in a FORTRAN program causes the program to perform input and output through the terminal.

The easiest way to cause a program to write data to the terminal is to specify an asterisk (\*) for the unit specifier in a WRITE or PRINT statement. This causes the output file to default to \$OUTPUT. For example, the following statement displays the string TYPE INPUT VALUES at the terminal:

```
PRINT *, 'TYPE INPUT VALUES'
```

An alternate way of writing data to the terminal is to associate a unit number with file \$OUTPUT in an OPEN statement. Then, when you reference the unit number in an output statement, data is written to file \$OUTPUT. For example, the following statements use this method to display two values at the terminal:

```
OPEN (UNIT=5, FILE='$OUTPUT')
WRITE (UNIT=5, FMT=12) R, ANS <-- Displays the values of R and ANS
```

The easiest way of reading data from the terminal is to specify an asterisk for the unit specifier in a READ statement. This causes the input file to default to file \$INPUT. For example, the following statements read two values from the terminal:

```
READ (UNIT=*, FMT=100) X, Y
100 FORMAT (2F6.2)
```

When the READ statement is executed, the system displays a question mark and the program pauses until you enter values for X and Y.

An alternate way of reading data from the terminal is to associate a unit number with file \$INPUT in an OPEN statement. Then, when you reference the unit number in a READ statement, data is read from file \$INPUT. For example, the following statements perform the same operation as the preceding example:

```
OPEN (UNIT=2, FILE=$INPUT)
READ (UNIT=2, FMT=100) X, Y
100 FORMAT (2F6.2)
```

When the READ statement is executed, execution pauses until you enter values for X and Y.

In each of the preceding examples, when the READ statement is executed, the system displays a question mark and waits for you to enter the input values. If you press RETURN before entering enough input values to satisfy the input/output list of the read statement, the system issues another ? prompt, and will continue to do so until the input/output list is satisfied.

## INTERACTIVE INPUT AND OUTPUT

For example, when the following READ statement is executed:

```
      READ (UNIT=*, FMT=12) X, Y
12  FORMAT (2f6.2)
```

the system displays a ? and waits for your input. If you type a single number, such as

```
? 123.45
```

the system will issue another ? prompt and wait for you to type another number. When the input/output list is satisfied, execution continues with the next statement.

If you do not know the exact number of input records your program will read, you will need to include a test for the end of the input data.

When reading data from a noninteractive file, you can use the END= specifier in the READ statement to test for end of data. However, the END= specifier can't be used for interactive input. Therefore, you must devise your own test for end of data. For example, the following sequence reads input records until the value 999.99 is read for both X and Y. The loop is then exited.

```
      READ (UNIT=*, FMT=12) X, Y
12  FORMAT (2F6.2)
      IF (X .EQ. 999.99 .AND. Y .EQ. 999.9) GO TO 15
      .
      .
      GO TO 10
15  CONTINUE
```

Whenever you write a program that reads data from the terminal, you can improve the usability of the program by including a statement to display an informative prompt immediately before the READ statement.

For example, when the following statements are executed:

```
      PRINT *, ' ENTER YOUR NAME '
      READ (UNIT=*, FMT=20) NAME
22  FORMAT (A20)
```

the following display appears at the terminal:

```
ENTER YOUR NAME
?
```

You then type a sequence of characters and press RETURN; the program then reads the characters and continues executing.

The second way of causing a program to receive input and display output at the terminal is to explicitly associate a file with the terminal. Then, any data written to that file is displayed at the terminal, and any data read from the file must be entered through the terminal.

You can associate a file with the terminal by specifying a `REQUEST_TERMINAL` command. This command has the form

```
REQUEST_TERMINAL FILE=file
```

You can associate a file with the terminal from inside a FORTRAN program with the statement

```
CALL CONNEC (u)
```

where `u` is the unit number of the file to be associated with the terminal.

Any data written to the file specified in a `REQUEST_TERMINAL` command or `CALL CONNEC` statement is displayed at the terminal, and any data read from the file must be entered through the terminal.

The file remains associated with the terminal until a `CALL DISCON (u)` statement is executed, or until you either enter a `DETACH_FILE` command or end the terminal session.

For example, suppose a program contains the following statements:

```
OPEN (UNIT=3, FILE='WHAT_FILE')  
READ (UNIT=3, FMT=100) (A(I),I=1,100)
```

These statements open a file named `WHAT_FILE` and read values into an array from that file. Normally, the system would look for `WHAT_FILE` in the `$LOCAL` catalog. But suppose you want the program to accept input directly from the terminal. Before executing the program, enter the command

```
REQUEST_TERMINAL FILE=WHAT_FILE
```

This command associates the file name `WHAT_FILE` with the terminal. When the `READ` statement is executed, the system will issue a `?` prompt and wait for you to enter values for array `A`.

# Copying Files Between NOS and NOS/VE

You can transfer files directly between the NOS and NOS/VE systems on the dual-state computer.

This discussion deals only with the transfer of files consisting of display code data. This includes files such as FORTRAN source programs and files used for formatted, list directed, and namelist input/output. The transfer of files containing other types of data (for example, files used for unformatted, indexed sequential, or direct access input/output) must be done using other tools. (Refer to the Migration from NOS to NOS/VE manual for additional migration information.)

You can copy either an indirect access or a direct access file from the NOS system to the NOS/VE system by using the GET\_FILE command. This command has the form

**GET\_FILE FROM=nos-file TO=nosve-file DC=64**

This command copies the specified direct or indirect access NOS file to the specified NOS/VE file. The NOS/VE file can be either a local file or a permanent file with the file name prefixed by the catalog names.

The NOS file is assumed to contain character data in 6/12 bit display code format. (You can use the GET\_FILE command to convert data in other formats. However, most NOS text files use the 6/12 display code format. Refer to the Migration from NOS to NOS/VE manual for more information about migrating files.)

For example, suppose you have a source program stored on a NOS permanent file named NOSPROG. You can copy that file to a NOS/VE permanent file named VEPROG as follows:

**GET\_FILE FROM=NOSPROG TO=\$USER.VEPROG**

This command copies the contents of the NOS file NOSPROG to the NOS/VE file VEPROG that resides in the master catalog. If VEPROG does not already exist, it is created.

Note that when you bring a FORTRAN source program from NOS to NOS/VE, you may have to change the program before you can execute it. For more information on how to migrate programs from NOS to NOS/VE, refer to the Migration from NOS to NOS/VE manual.



You can copy a NOS/VE file to a NOS file by using the command

```
REPLACE_FILE TO=nos-file FROM=nosve-file
```

Each ASCII character in the NOS/VE file is converted to a 6/12 bit display code representation.

If the NOS file you specify already exists on the NOS system (as a direct or indirect access permanent file), it is replaced by the specified NOS/VE file. If the NOS file does not already exist, a new direct access permanent file is created and the NOS/VE file is copied to it.

For example, assume you have a local NOS/VE file named VEFIL. You can copy that file to the NOS side of the computer by entering

```
REPLACE_FILE TO=NOSFIL FROM=VEFIL
```

File NOSFIL now exists as a permanent file under NOS.

Additional parameters on the REPLACE\_FILE command provide other capabilities. For example, a parameter is provided that enables you to copy NOS files belonging to another user, and another parameter enables you to copy NOS files having formats other than 6/12 display code. These parameters are described in the SCL Language Definition manual.

## **Compiling and Executing a FORTRAN Program**

NOS/VE provides commands for compiling and executing FORTRAN programs. The following paragraphs describe the most frequently used parameters and options.

### **Compiling**

The FORTRAN compiler reads a source program and generates an executable object program. The compiler provides a number of options, including names of the input and output files, level of optimization, and output listing options.

The compiler is called by the FORTRAN command. The FORTRAN command parameters are described in detail in the FORTRAN Language Definition manual. This discussion presents a brief overview of the FORTRAN command and some suggestions for selecting various options.

For most compilations, you will need to specify no more than two or three parameters on the FORTRAN command. These parameters specify the name of the file containing the source program, the file to receive the executable object program, and the file to receive the compiler output listing (along with error messages).

FORTRAN also provides a parameter that allows you to specify a file to receive compile-time error messages. If you omit this parameter, the compiler writes error messages to the terminal and to the output listing file.

For example, the command

```
FORTRAN I=SRCE B=BIN L=FLIST
```

reads source input from file SRCE, writes object code to file BIN, and writes the output source listing to file FLIST. These files are all local files.

To specify permanent files, you can prefix the file names with catalog names. For example, in the command

```
FORTRAN I=$USER.SRCE B=$USER.BIN L=$USER.FLIST
```

files SRCE, BIN, and FLIST are permanent files in the master catalog.

If you omit any of these parameters, a default value is provided. For the I parameter, the default is \$INPUT, which cannot be used for compiling at the terminal. For the B parameter, the default is \$LOCAL.LGO. If you omit the L parameter, no source listing is produced at the terminal, but a printed listing is produced in batch mode.

During debugging, you may want to generate a reference map, a useful debugging tool. The parameter to generate a complete reference map is LO=(R,A,M). An explanation of how to use the reference map is presented in chapter 2, Debugging.

The preceding parameters are the most commonly used of the FORTRAN parameters. Refer to the FORTRAN Language Definition manual for complete descriptions of all the FORTRAN command parameters.

## Executing

You can begin execution of a FORTRAN object program in two ways. The simplest and most common way is use a name call command. To use this command, you simply specify the name of the file containing the object program. (This is the file specified by the BINARY\_OBJECT parameter on the FORTRAN command.) For example, the following commands compile and execute a FORTRAN program:

```
FORTRAN I=SRCE B=BIN L=LISTFIL  
BIN
```

The FORTRAN command compiles the source program on file SRCE and writes object code to file BIN. The BIN command begins execution of the object program.

The following example is equivalent to the preceding one, except the common practice of writing the object program to the default file LGO is used:

```
FORTRAN I=SRCE L=LISTFIL
LGO
```

You can specify an optional list of parameters on the name call command. These parameters fall into the following three classes:

- The \$PRINT\_LIMIT and STATUS parameters. The \$PRINT\_LIMIT parameter specifies the maximum number of lines that an executing program can write to file \$OUTPUT. The STATUS parameter returns an SCL error code when the program terminates.
- The file name substitution parameters. These parameters enable you to specify file names on the execution command that are substituted for file names in the program.
- The user-defined SCL parameters. These parameters enable you to pass values to a program by specifying them on the execution command.

Refer to the FORTRAN Language Definition manual for detailed descriptions of these parameters.

The second way of beginning execution of a compiled FORTRAN program is through the EXECUTE\_TASK command. This command has the same effect as the name call command, but it allows you to specify a number of additional options. The EXECUTE\_TASK command has the form

#### **EXECUTE\_TASK parameter-list**

The parameter list enables you to select options that affect execution of your program. The simplest form of this command is the same as a name call command. For example, the following commands are equivalent:

```
EXECUTE_TASK FILE=LGO

LGO
```

Both commands begin execution of the object program on file LGO.

Following are discussions of some commonly-used EXECUTE\_TASK options.

## Loading Modules From Different Files

When you execute a program using the name call command, the system assumes that all program units required for execution are contained in the object file you specify (or in libraries available to the program). For example, if you have a main program PROGA and a subroutine SUBA, both PROGA and SUBA must reside on the file you specify in the name call command. In general, if you write a main program and subroutines all at once, you place them on the same file.

However, suppose your program calls a subroutine that resides on a different file. If you try to use the name call command, the system will issue a runtime error when the program calls the subroutine. But you can use the EXECUTE\_TASK command to cause the desired subroutine to be included in the executable program.

There are two EXECUTE\_TASK parameters that you can use to include program units from other files in your program. The parameter

```
FILE=(file-list)
```

causes the system to include all the program units on all the specified files in your program. For example, suppose a program on file LGO calls subroutines on files SUBFILE1 and SUBFILE2. The command

```
EXECUTE_TASK FILE=(LGO, SUBFILE1, SUBFILE2)
```

causes program units on files LGO, SUBFILE1, and SUBFILE2 to be included in the executable program. Execution begins with the first file in the list, in this case, LGO.

## Loading Modules From Libraries

A library is a file that contains object programs in a special format that allows rapid searching and loading. If you have program units on libraries, such as those created by the CREATE\_OBJECT\_LIBRARY utility, you can direct the system to search those libraries by using the EXECUTE\_TASK command. The parameter that specifies libraries to be searched has the form

```
LIBRARY=(lib,...,lib)
```

where lib is a library file to be included in the search. The system searches the specified libraries, in the order listed, for any program units required by the program, and includes those program units in the executable program.

More information about creating and using object libraries is presented in chapter 6, Using Object Libraries.

## Requesting a Load Map

The EXECUTE\_TASK command allows you to request a load map and to write it to a separate file. The load map contains storage allocation information about your program. It can be a useful debugging tool although, in most cases, you need thorough knowledge of how NOS/VE allocates storage is required in order to interpret a load map.

You request a load map by specifying the LOAD\_MAP\_OPTION parameter on the EXECUTE\_TASK command. The load map consists of several different parts, each of which is controlled by an option on the LOAD\_MAP\_OPTION parameter. The following EXECUTE\_TASK command begins execution of the program on file LGO and generates a complete load map:

```
EXECUTE_TASK FILE=LGO LOAD_MAP_OPTION=(S,B,EP,R)
```

For a detailed description of the information that is included in the map, refer to the SCL Object Code Management manual.

## Presetting Memory

The EXECUTE\_TASK command provides an option that enables you to select a memory preset value. This is a value to which all unused locations in your program's area are set before execution begins. Presetting memory to a particular value can help you debug a program.

The memory preset value is controlled by the PRESET\_VALUE parameter on the EXECUTE\_TASK command. You can select several different values. The following example presets memory to floating-point infinite before beginning execution of the program on file LGO:

```
EXECUTE_TASK FILE=LGO PRESET_VALUE=INFINITY
```

More information on how this option can help you debug a program is presented in chapter 6, Debugging.

## Submitting Batch Jobs

NOS/VE allows you to easily submit batch jobs from a terminal.

The difference between a batch job and an interactive terminal session is that in the terminal session, each command you enter is executed immediately after you enter it. An interactive terminal session consists of a sequence of interactions between you and the computer; you enter system commands and the computer responds to each command as it is entered.

In batch execution, you submit a complete sequence of commands (usually contained in a file) to the computer and the computer executes those commands without any communication or intervention from the terminal.

The advantage of submitting a batch job from the terminal is that while the job is executing, the terminal is free for other uses. Thus, batch execution is especially useful for time-consuming operations that would otherwise occupy a terminal for long periods of time.

For example, suppose you want to compile a long FORTRAN program. If you initiate the compilation interactively, the terminal may be tied up for a long period of time. However, if you submit the compilation as a batch job, the terminal remains free while the program is being compiled.

A second advantage of submitting batch jobs is that it is sometimes easier to submit long sequences of commands that will be executed repeatedly as a batch job, rather than repeatedly entering the sequence interactively each time you want to execute it.

To submit a batch job from the terminal, you must first create a file that contains the commands to be in the job. You then submit the job by specifying a `SUBMIT_JOB` command.

The first step in preparing a batch job at the terminal is to create a file that contains the job. This file is simply a text file containing the commands to be executed. The first line of this file must be a `LOGIN` command. The `LOGIN` command has the form

**LOGIN USER=user-name PASSWORD=word FAMILY=  
family-name**

`NOS/VE` requires this command in order to validate the batch job. Typically, you will use the same information you used to log in at the terminal.

Following the `LOGIN` command are the commands to be executed in the batch job. A `LOGOUT` command at the end of the job is optional; when the last command is executed, the job simply terminates.

You can create the file containing the batch job using a text editor such as the `SCU` editor. However, the simplest way to create the file is to use the `COLLECT_TEXT` command.

The `COLLECT_TEXT` command provides an easy way for you create a text file at the terminal. This command has the form

**COLLECT\_TEXT FILE=file-name**

When you enter a COLLECT\_TEXT command, the system responds with the prompt

```
ct?
```

and waits for you to enter a line of text. After you enter a line of text and press RETURN, the system issues another ct? prompt. The system continues to issue ct? prompts in response to your entries until you enter the string \*\*. This signals the end of the sequence of text lines and terminates the COLLECT\_TEXT command. The text lines you entered are then written to the file you specified in the COLLECT\_TEXT command.

For example, suppose you have a FORTRAN source program on file FORT\_PROG, and you want to compile that program in a batch job. The following COLLECT\_TEXT command creates a file named BATCH\_COMP that contains commands to create a permanent file for the object program and to compile the source program:

```
/collect_text output=batch_comp
ct? login user=username password=urpass family_name=urfam
ct? create_file $user.fort_bin
ct? fortran i=fort_src b=$user.fort_bin
ct? **
```

File BATCH\_COMP, containing the batch job, now exists as a local file.

To submit the batch job to NOS/VE, you use the SUBMIT\_JOB command. This command has the form:

```
SUBMIT_JOB FILE=file JOB_NAME=name
```

where file is the file containing the batch job, and name is an arbitrary name that you assign to the job. For example, the command to submit the preceding job is

```
SUBMIT_JOB FILE=BATCH_COMP JOB_NAME=MYJOB
```

BATCH\_COMP is the file containing the batch job and MYJOB is a name we assign to the job.

After you have submitted a batch job, you will probably want to check periodically to find out if the job has completed. You can check the progress of a batch job by entering a `DISPLAY_JOB_STATUS` command. This command has the form

**DISPLAY\_JOB\_STATUS JOB\_NAME=name**

where name is the name you assigned to the job in the `SUBMIT_JOB` command. This command displays a short paragraph of information about the specified job. Included in this information is the line:

**JOB\_STATE:** string

where string indicates the current status of the job. The displayed string tells you whether the job has been initiated and, if so, whether it is executing or waiting for a system resource, such as memory space or a permanent file, to become available.

If the system responds to a `DISPLAY_JOB_STATUS` command with the message

**NAME NOT FOUND:** job-name

the job has probably completed.

To display the status of our sample job, the command is

**DISPLAY\_JOB\_STATUS JOB\_NAME=MYJOB**

If, in the paragraph of information displayed by the system, the following appears:

**JOB\_STATE:** EXECUTE

we know that the job is executing. If the following appears:

**NAME NOT FOUND:** MYJOB

we know that the job has completed.



## Executing SCL Commands Inside a FORTRAN Program

NOS/VE FORTRAN allows you to execute SCL commands within a program. This is accomplished through the SCLCMD call. The SCLCMD call has the form

```
CALL SCLCMD ('scl-command')
```

where scl-command is any valid SCL command. When the SCLCMD call is executed, the specified SCL command is passed to SCL and executed.

The SCLCMD call is especially useful for creating and retrieving permanent files. A FORTRAN application program that has permanent file commands embedded within the program is easier to use than one that requires a user to enter the commands outside of the program.

As an example, the following program attaches and reads an existing file, and creates and writes a new file. The input file is named INFILE and the output file is named OUTFILE. The program uses the SCLCMD call to execute the necessary SCL commands.

```
PROGRAM RDWRT
DIMENSION A(100), B(100)
CALL SCLCMD ('ATTACH_FILE $USER.INFILE') <--- Make input file
OPEN (UNIT=2, FILE='INFILE')                available to job.
READ (UNIT=2) A
CALL SCLCMD ('DETACH_FILE INFILE') <----- Detach input file
                                           from job.
-
-
-
CALL SCLCMD ('CREATE_FILE $USER.OUTFILE') <-- Create permanent
                                           file to receive
                                           output.
CALL SCLCMD ('ATTACH_FILE $USER.OUTFILE') <-- Make output file
OPEN (UNIT=3, FILE='OUTFILE')                available to job.
WRITE (UNIT=3) B
CALL SCLCMD ('DETACH_FILE OUTFILE') <----- Detach output file
END                                           from job.
```

# Passing Parameters to a FORTRAN Program

NOS/VE FORTRAN enables you to pass parameters between the system and a FORTRAN program. You specify the parameters on the execution command that begins execution of the program.

This parameter interface capability is useful in the design of applications that require users to select certain options or to specify values used by the program. For example, using the SCL parameter passing capability, you could design a FORTRAN program that would allow a user to specify the names of the input and output files.

Consider the following simple example of a program that reads data from a file and writes the same data to another file:

```
PROGRAM SRCE
DIMENSION A(100)
OPEN (UNIT=2, FILE='INDAT')
OPEN (UNIT=3, FILE='OUTDAT')
READ (UNIT=2) A
WRITE (UNIT=3) A
END
```

This program reads from a file named INDAT and writes to a file named OUTDAT. Now, suppose we want to write a more general program that will allow a user to pass the file names as parameters to the program.

In other words, we want to design a program whose execution command looks something like this (assuming that APPL is the name of the file containing the object program):

```
APPL INPUT_FILE=file-1 OUTPUT_FILE=file-2
```

where file-1 and file-2 are file names supplied by the user of the program. (The names INPUT\_FILE and OUTPUT\_FILE are names of our own choosing.) Then, a user of program APPL could supply his own file names. For example, the user might specify

```
APPL INPUT_FILE=AAA OUTPUT_FILE=BBB
```

where AAA and BBB are file names.

We now show how you can use the SCL parameter interface capability to design a program that accepts parameters specified on the execution command.

Consider the command from the preceding example:

```
APPL INPUT_FILE=file-1 OUTPUT_FILE=file-2
```

This command assumes that we have compiled a FORTRAN program and written the object program to a file named APPL. But before we can specify parameters on the execution command, we must make certain changes to the source program. First, we examine at the components of an SCL parameter.

An SCL parameter consists of two parts: a name and a value.

```
INPUT_FILE=file-1
```

The first step is to define the parameter name and value in the source program, so that the program has all the necessary information about the parameters.

You define SCL parameters by including a C\$ PARAM directive in the program. This directive has the form

```
C$    PARAM ('pdef, . . ., pdef')
```

where pdef is a parameter definition. The parameter definitions include such information as the parameter name, the type (or SCL kind) of the parameter value, and the parameter default value (if any). The parameter definitions follow a strict set of rules defined by SCL; these definitions can get somewhat involved. For a complete description of how to define SCL parameters, refer to the SCL Language Definition manual. For purposes of the following example, some simple parameter definitions will suffice.

We want to design a program that will accept parameters specified on the execution command as follows:

```
APPL INPUT_FILE=file1 OUTPUT_FILE=file2
```

The first step is to define the parameters by including the following C\$ PARAM directive in the program:

```
C$    PARAM ('INPUT_FILE:FILE; OUTPUT_FILE:FILE')
```

This directive tells the FORTRAN compiler that the execution command will contain two parameters. The parameter names are INPUT\_FILE and OUTPUT\_FILE. The string :FILE after the parameter name specifies that the parameter is of type FILE; that is, values specified for the parameters on the execution command will be file names.

After defining the parameters with a C\$ PARAM directive, the next step is to provide for retrieving the parameters in the program. This is accomplished through the parameter interface subprograms.

The parameter interface subprograms provide the mechanism through which you retrieve parameter values during execution of your program. A separate call is provided for each of the different parameter types. The call we must use in our example to retrieve the FILE parameters is the GETCVAL call. In this discussion, we do not give the format of the GETCVAL call, because it is rather long. Instead, we just show the call used in the example.

Since our example has two parameters, the program must have two GETCVAL calls. The first one is:

```
CALL GETCVAL ('INPUT_FILE', 1, 1, 'LOW', LEN1, FNAME1)
```

The first argument, 'INPUT\_FILE', specifies the parameter name. The fifth argument, LEN1, is a variable in which the length (in characters) of the parameter value will be returned. The sixth argument, FNAME1, is a character variable in which the actual value of the parameter will be returned. (The variable names, of course, are of our own choosing.) The remaining arguments are not used in this example, but they must be assigned values anyway.

Similarly, the call to retrieve the second parameter is:

```
CALL GETCVAL ('OUTPUT_FILE', 1, 1, 'LOW', LEN2, FNAME2)
```

We have now defined our parameters in a C\$ directive and written GETCVAL calls to retrieve the parameter values during execution. We can put all this together in the following example program:

```

PROGRAM SRCE
C
C Define parameters INPUT_FILE and OUTPUT_FILE.
C$  PARAM ('INPUT_FILE:FILE; OUTPUT_FILE:FILE')
C
      DIMENSION A(100)
      CHARACTER*31 FNAME1, FNAME2
C
C Retrieve parameter values.
      CALL GETCVAL ('INPUT_FILE', 1, 1, 'LOW', LEN1, FNAME1)
      CALL GETCVAL ('OUTPUT_FILE', 1, 1, 'LOW', LEN2, FNAME2)
C
      F1 = 'ATTACH_FILE $USER.'//FNAME1)
      F2 = 'ATTACH_FILE $USER.'//FNAME2)
C
C Attach permanent files.
      CALL SCLCMD (F1)
      CALL SCLCMD (F2)
C
C Use parameter values in OPEN statements.
      OPEN (UNIT=2, FILE=FNAME1)
      OPEN (UNIT=3, FILE=FNAME2)
C
      READ (UNIT=2) A
      WRITE (UNIT=3) A
      END

```

Note that in the CHARACTER statement, we have allowed 31 characters for the file name, because that is the maximum length of a NOS/VE file name.

We now have a generalized FORTRAN program for which a user can specify the names of the input and output files on the execution command. Assuming the source program is on file SPROG, we can compile the program with the following command:

```
FORTRAN I=SPROG B=APPL
```

An example of a command to execute the program is:

```
APPL INPUT_FILE=MYFILE OUTPUT_FILE=URFILE
```

The program reads input from MYFILE and writes output to file URFILE.

## Short Forms of SCL Commands

Every SCL command has both a long form and a short form. The long form can be useful for beginners because the command name and parameter names consist of words that are descriptive of what the command does. However, the short forms are much easier to type, and you are encouraged to use them as you become more familiar with the system.

The short forms are derived from the long forms according to specific rules. Thus, if you know the long form of any SCL command, you can apply the rules to determine the short form.

Most SCL commands consist of either two or three words. To see how the short forms are derived, we'll consider an example of a two-word command and of a three-word command.

An example of a two-word command is the COPY\_FILE command. To derive the short form, join the first three letters of the first word and the first letter of the second word. For example, the short form of COPY\_FILE is COPF. Following are some additional examples of two-word commands, showing the long and short forms:

```
CREATE_FILE   CREF
ATTACH_FILE   ATTF
GET_FILE      GETF
```

To derive the short form of a three-word command, join the first three letters of the first word with the first letter of the second word and the first letter of the third word. For example, the short form of SET\_FILE\_ATTRIBUTE is SETFA. Following are additional examples of three-word commands:

```
CREATE_FILE_CONNECTION  CREFC
SET_PROGRAM_ATTRIBUTE    SETPA
DISPLAY_FILE_ATTRIBUTE   DISPA
```

The parameters for most SCL commands can be abbreviated by joining the first letter of each word of the parameter name. For example, in the command

```
FORTRAN INPUT=S LIST_OPTIONS=(S,A)
```

the parameters can be shortened to

```
FORTRAN I=S LO=(S,A)
```

Note that there are a few commands and parameters that do not conform to the rules for abbreviating commands. For example, the FORTRAN command is abbreviated FTN, and the BINARY\_OBJECT parameter is abbreviated B. For these commands, you must simply memorize the short form.

## Summary of NOS/VE Capabilities

This chapter introduced you to some of the commands and capabilities of NOS/VE. The following summary highlights the key topics discussed in this chapter.

### Files

Under NOS/VE, all of your programs and data are stored in files. A file is a collection of information that begins at a boundary called the beginning-of-information and ends at a boundary called the end-of-information.

All files are either local files or permanent files. Local files do not exist beyond the end of a terminal session. Permanent files continue to exist after the end of a session and can be accessed in subsequent sessions.

All files reside in catalogs. A catalog is a collection of files identified by a catalog name. All users have two catalogs: the \$LOCAL catalog, which contains all of your local files, and the \$USER catalog, which contains all of your permanent files.

You can create subcatalogs within the \$USER catalog. A subcatalog is simply a catalog within a catalog. Subcatalogs provide a way for you to organize and keep track of your permanent files.

You reference a file by specifying the file name and the catalog in which it resides. Examples of file references are:

\$LOCAL.NEW_FILE	References file NEW_FILE in the \$LOCAL catalog.
------------------	--

\$USER.CAT1.DATA_FIL	References file DATA_FIL in subcatalog CAT1 of the \$USER catalog.
----------------------	--

If you omit the catalog name, the system looks for the file in the current working catalog. The default working catalog is \$LOCAL; however, you can change the working catalog to one of your permanent file catalogs.

Remember, though, that files referenced in a FORTRAN program must reside in the \$LOCAL catalog, even if you change the working catalog.

Normally, other users cannot access your permanent files. However, the CREFP command enables you to give other users permission to access your files.

NOS/VE provides commands that enable you to manage your local and permanent files. Following is a summary of these commands.

<u>Operation</u>	<u>Command</u>
Create a file	CREATE_FILE
Make a file available to a terminal session.	ATTACH_FILE
Delete a local file	DETACH_FILE
Delete a permanent file	DELETE_FILE
Copy one file to another	COPY_FILE
Create a catalog	CREATE_CATALOG
Display a list of files in a catalog	DISPLAY_CATALOG
Delete a catalog	DELETE_CATALOG
Change the working catalog	SET_WORKING_CATALOG
Change the access permission of a file	CREATE_FILE_PERMIT

## Terminal Input and Output

FORTRAN provides the capability of writing programs that display output at the terminal and read input from the terminal.

The most common way of performing interactive input and output is to substitute an asterisk (\*) for the unit specifier in a READ or WRITE statement. This causes the program to read from the standard system file \$INPUT and write to the standard system file \$OUTPUT.

Files \$INPUT and \$OUTPUT are automatically associated with the terminal. Thus, a program that reads from file \$INPUT expects data to be input from the terminal, and a program that writes to file \$OUTPUT displays output at the terminal.

You can also use the REQUEST\_TERMINAL command to associate a file with the terminal. Then, all reading and writing to that file is performed through the terminal.



## NOS to NOS/VE File Transfer

NOS/VE provides commands that enable you to copy text files from NOS/VE to NOS and from NOS to NOS/VE.

The command to copy a file from NOS to NOS/VE is

**GET\_FILE FROM=nos-file TO=nosve-file**

The command to copy a file from NOS/VE to NOS is

**REPLACE\_FILE FROM=nosve-file TO=nos-file**

## Compiling and Executing

NOS/VE provides commands for compiling and executing FORTRAN source programs.

The command to compile a FORTRAN program is:

**FORTRAN parameter-list**

where parameter-list selects compiler options. The following FORTRAN command is typical:

**FORTRAN I=source-input B=object-output L=list-file**

The I parameter specifies the file containing the source program, the B parameter specifies the file to receive the object code, and the L parameter specifies the file to receive the source listing.

You can begin execution of a compiled program in two ways. The simplest way is to specify a name call command. This command consists of the name of the file containing the object code. For example,

LGO

begins execution of the program on file LGO.

The second way of beginning execution is through the EXECUTE\_TASK command. This command has the same effect as the name call command, but it provides more options. These options include generating a load map, loading programs from separate files, and specifying libraries to be searched before the program is executed. An example of an EXECUTE\_TASK command is

**EXECUTE\_TASK (LGO, FILE1, FILE2)**

The system first loads programs from files LGO, FILE1, and FILE2 into a single executable module, and begins execution of the program on LGO.

You can create and submit batch jobs while executing at the terminal. The first command of the batch job must be a LOGIN command. The commands to be included in the job follow the LOGIN command.

The job must be contained in a text file. You can create the file using an editor such as the SCU editor, or through the COLLECT\_TEXT command. The COLLECT\_TEXT command provides an easy way of entering lines into a text file.

You submit the batch job to the system by specifying a SUBMIT\_JOB command.

After you have submitted a batch job, you can check the progress of the job by specifying a DISPLAY\_JOB\_STATUS command. This command tells you, among other things, whether the job has completed.

## Passing Parameters

NOS/VE allows you to pass parameters to a FORTRAN program by specifying the parameters on the execution command. This capability is useful for designing applications where the user can specify values or request options.

The first step in designing a program that can receive parameters on the execution command is to define the parameters. This is accomplished through the C\$ PARAM directive. This directive is placed anywhere in the source program. The C\$ PARAM directive specifies a complete description of each parameter, including the parameter name and type.

The next step is to include one or more calls to the parameter interface subprograms in the program. These calls retrieve the values of the parameters specified on the execution command. You need one call for each parameter. A different call is provided for each of the parameter types. The parameter interface subprograms are described in the FORTRAN Language Definition manual.

An important step in the development of any program is the process of detecting and correcting all the errors in the program. Several tools are provided to assist you in debugging a program.

## The Debugging Process

The debugging process actually begins when you write a program. If you write error-free code to begin with, you do not need to spend time debugging. Of course, few people write error-free code. But there are some programming techniques you can apply that will minimize the errors in a program. And when your program does have errors, there are some options and utilities that can help you locate those errors.

## How Errors Affect a Program

No matter how careful you are when you write a program, the program is likely to contain at least a few errors. Program errors fall into three classes, according to how they affect your program. The three classes of errors are:

Errors that prevent the program from compiling. These errors result in a compiler-generated error message. Since those messages are considered to be self-explanatory, compile-time debugging is not discussed here.

Errors that cause the program to terminate prematurely. These errors result in an error message that may or may not pinpoint the error for you.

Errors that allow the program to run to completion, but result in incorrect output. These errors are often the most difficult to find because there is generally no indication as to where the error occurred or what caused the error.

## Before Executing Your Program

Since computer time is an expensive resource, you should try to correct as many errors as possible before you execute your program. Before compiling, carefully review the program for the more obvious mistakes. But no matter how careful you are, you may still have to compile several times before the program is ready to execute.

When you compile your program for debugging, there are some options you should select (or, in some cases, not select) on the FORTRAN command:

- Specify `OPTIMIZATION=DEBUG` (or omit this parameter altogether, since `DEBUG` is the default option). This ensures the fastest possible compilation time. During the debugging stages, fast compilation is generally more important than fast execution.

If you do intend to run optimized object code, however, you should do a final test run under `OPTIMIZATION=HIGH` after your program has been completely debugged under `OPTIMIZATION=LOW`. This is advisable because, in rare cases, the optimization process can alter the program's results.

- Specify `LIST_OPTIONS=(S,A,R,M)`. This generates a complete reference map. The reference map is an extremely useful debugging tool that provides concise descriptions of all the symbolic names used in the program. When you compile your program during the debugging phase, you should always request a reference map and examine it carefully. This step can detect many errors that would be much more difficult to find otherwise. More on the reference map later.
- Specify `LIST=file-name`. The `LIST` parameter specifies the file to receive the compiler output listing (including the reference map). This is especially important for interactive jobs, because if you omit this parameter, output is written to file `$NULL` (in which case the output is lost).

The following FORTRAN parameters can be useful in certain situations:

- `ERROR=file-name`. This parameter specifies a file to receive all compiler-generated error messages. Writing error messages to a separate file can be useful for longer programs that might produce a large number of messages.
- `BINARY_OBJECT=$NULL`. Writing the object code to file `$NULL` has the effect of discarding the code. While you're in the process of correcting compile-time errors, you can save storage space by selecting this option.

After you have compiled your program successfully and examined the reference map, you are ready to begin executing the program. But first, there are a couple of system commands that can make execution-time debugging easier.

Before executing your program, you should set message mode to `FULL`. The command is

```
SET_MESSAGE_MODE FULL
```

Setting message mode to **FULL** causes the system to display additional information whenever an execution error occurs. The exact nature of this information will be discussed later.

Before executing your program, you should ensure that the preset value (the value to which all uninitialized memory locations are set) is either floating-point infinite or floating-point indefinite. Either of these values will cause an error message and abort when the value is used in a floating-point calculation. This makes it easier to detect the common error of using undefined variables in expressions. You can find out what the current memory preset is by entering the command

```
DISPLAY_PROGRAM_ATTRIBUTES
```

If memory is preset to zero (a common default), errors caused by use of undefined variables can be much more difficult to detect. You can set the memory preset value through the following commands:

```
SET_PROGRAM_ATTRIBUTE PRESET_VALUE=INDEFINITE
```

Sets all uninitialized locations to floating-point indefinite.

```
SET_PROGRAM_ATTRIBUTE PRESET_VALUE=INFINITY
```

Sets all uninitialized locations to floating-point infinite.

## What Should You Do When an Error Occurs?

Some programming errors will cause your program to terminate prematurely and produce an execution-time error message. Other errors allow the program to run to completion but cause incorrect results.

After you receive an error message, the first step is usually to examine the job log. The job log is a chronological history of events leading up to the execution of your program. The job log shows all the commands you entered and any error messages generated by the commands. This information can sometimes be helpful in detecting errors. The command to display the job log is

```
DISPLAY_LOG
```

Sometimes the error message and job log will give you enough information to find the error. If not, the next step is to search for more information about the message. The best source of information is the online manual **MESSAGES**, which provides explanations of all runtime error messages issued under **NOS/VE**. To find the description of a particular message, you can use one of the commands **HELP** or **EXPLAIN\_MESSAGE**. These commands are discussed later in this chapter.

After you have investigated the error message and read the explanations, you may still need to determine exactly where in your program the error occurred and what caused the error. You can proceed in several ways.

First, a visual check of your program might reveal some errors. Some common errors to look for are discussed later in this chapter.

If a visual check does not reveal the errors, you can use one of the debugging aids available under NOS/VE to help locate errors in your program. The Debug facility, described later in this chapter, enables you to debug your program while it is executing. You can use Debug to stop execution at specified points, or on the occurrence of an error, and to display the contents of variables and arrays while execution is stopped.

## Common Runtime Error Conditions

Many runtime errors occur either when an arithmetic operation generates an invalid value or when an invalid value is used in an arithmetic operation. The causes of these errors can be difficult to locate because the error message is often not closely related to the actual programming error that caused the message to be issued.

When these errors occur, the best course of action usually is to first examine the program closely for logic errors, and then to use the Debug utility to determine how the invalid values were generated.

### Errors Caused by Indefinite Values

An indefinite value usually results when a floating-point calculation cannot be resolved, such as a division where the dividend and divisor are both zero.

You can also set the memory preset value to indefinite so that any undefined variables in the program will have an indefinite value. In this case, using an undefined variable in an expression will cause a runtime error.

An indefinite value has a specific internal representation that does not correspond to a number, and that causes an error when used in a calculation. Errors involving indefinite values are diagnosed during execution as follows:

The message "PM F. P. indefinite at {hex-address}" is issued. (F. P. stands for floating-point). This happens when an indefinite value is used in a calculation.

A message of the form "function-name (argument). Argument indefinite." is issued. This occurs when an indefinite value is passed to an intrinsic function. For example, ALOG(argument). Argument indefinite.

A WRITE or PRINT statement prints the string I or INDEFINITE. This happens when one of the variables in the input/output list contains an indefinite value.

If your program contains errors involving indefinite values, check for logic errors such as dividing zero by zero or use of undefined variables.

### Errors Caused by Infinite Values

An infinite value represents a floating-point number that exceeds the largest number that can be represented in floating-point format. An infinite value is generated when a calculation produces a mathematically infinite result. An infinite result often occurs when the operands of an expression contain values that are either very large or very small (approaching the limits of the computer), or are themselves infinite. For example, in the expression

$$1.0E+50 * 1.0E+50$$

both operands are legal values, but the result is an infinite value.

If the memory preset value is INFINITE (either by job default or previous SET\_PROGRAM\_ATTRIBUTE command), any undefined variables in the program will contain a floating-point infinite value. Use of these variables in an expression will generate an exponent overflow condition.

Infinite values in a program are diagnosed during execution in the following ways:

The message "PM exponent overflow at {hex-address}." is issued. This means that a calculation produced an infinite value.

A message of the form "function-name (argument). Argument infinite." is issued. This means that an infinite value was passed to an intrinsic function.

A message of the form "function-name (argument). Result infinite." is issued. This means that the indicated intrinsic function produced an infinite value.

A WRITE or PRINT statement prints the symbol R or INFINITE. The variable being printed contains an infinite value.

If your program contains errors involving infinite values, do the following:

Check the value of the memory preset. If it is INFINITE, check the program for undefined variables.

Check for operands that contain very large or very small values. Such values are often the result of bad input data, incorrect calculations (for example, a \* operation that was mistakenly typed as \*\*), or variables that contain the incorrect data type (make sure all your real variables contain floating-point values).

## Errors Involving Integer Arithmetic

Arithmetic overflow is the integer equivalent of an exponent overflow. It is caused when the result of an integer operation exceeds the largest value that can be represented in integer format. Arithmetic overflow is indicated by one of the following messages:

PM arithmetic overflow at {hex-address}.

This means that an expression generated an arithmetic overflow.

function-name (argument). Arithmetic overflow.

This means that the indicated intrinsic function (a type integer function) produced an arithmetic overflow.

Another common error involving integer arithmetic is indicated by the message

PM Arithmetic loss of significance at {hex-address}.

This message indicates that an operation on extremely large integer values caused truncation of the low-order digits of the integer result.

If an arithmetic overflow or loss of significance condition occurs, it generally means that the operands involved in the calculation, or passed to the function, contain incorrect values. Those values, when printed, contain as many as 19 digits. Check all previous uses of the operands to determine how the bad values were generated.

Arithmetic overflow can also be caused by referencing undefined integer variables. If the memory preset value is either indefinite or infinite, undefined integer variables will contain values that, while not themselves illegal, cause an arithmetic overflow condition when used in a calculation.



## Access Violation Error

Another message whose cause may not be obvious is

PM Access violation at (hexadecimal-address)

This message is issued when a program attempts to access a location outside of its assigned area. Common causes include

- A DO loop that causes a subscript to exceed its declared dimension bounds. In this case, the access violation message is usually preceded by a FORTRAN informative message of the form

Subscript {n1} of array {array-name} is {n2}. Declared lower bound is {n3}; upper bound is {n4}

- A function or subroutine call that contains more arguments than the function or subroutine dummy argument list.

## Divide Fault Error

The following error frequently occurs in arithmetic computations:

PM divide fault at (hexadecimal-address)

This message is issued when a program attempts to divide by zero, usually an indication of invalid data or incorrect logic. A good practice is to precede all statements in which zero division is a possibility by a test for a zero divisor.

## Common Programming Errors

Following are some common programming errors you can check for when debugging a program. Some of these errors cause the program to abort with a message such as those described above. Others allow the program to run to completion but cause incorrect results.

When a program runs to completion but produces incorrect results, the best course of action is to start at the beginning of the program and step your way through it to find out what went wrong. You can do this by using the Debug utility, described later in this chapter.

Following are some common programming errors to look for when debugging a program.

- Check for undefined or improperly initialized variables. An undefined variable is one that has not been stored into before being used in a computation. Using undefined variables gives unpredictable results.
- Check for misspelled or mistyped variable names. The reference map is extremely helpful in locating this type of error.
- Check for improper use of subscripts. In particular, check for DO loops that cause subscripts to exceed the declared dimension bounds. (A runtime check for dimension bounds violations is automatically performed.)
- Check for improper use of mixed mode arithmetic. For example, the calculation  $I = 3/2$  gives a different result than  $A = 3/2$ .
- Check for incorrect or invalid input data. Certain input values may cause errors in a logically correct program. In particular, check for very large or very small values and for values that would cause a division by zero.
- Check for accumulated roundoff error. This error will not usually produce an error message, but can affect the results. Roundoff error can occur when values used in calculations depend on previous calculations, or when very large and very small numbers are used in the same expression. A possible solution is to use double precision arithmetic.
- Check subroutine and function references to ensure that the actual arguments in the CALL statement or function reference agree in number and data type with the dummy arguments in the SUBROUTINE or FUNCTION statement.
- Check for incorrect logic. If the program contains no other errors, it could be that the program was simply designed wrong. Make sure that the algorithm is correct and that all the computations are correct.

If an execution error occurs during an input or output operation, there are some common causes you can check for.

- Do not mix I/O types on the same file. For example, you cannot perform direct access I/O on a file that was created as a sequential access file.
- When reading a file, always include the END= specifier in the READ statement to test for the end-of-information. An attempt to read beyond end-of-information causes an error.

- When reading formatted data, be sure that the data fields are consistent with the fields described in the `FORMAT` statement. Remember that a decimal point in an input field takes precedence over the position of the decimal point implied by a floating-point edit descriptor.
- In direct access input/output, be sure that you have specified the record length correctly in the `OPEN` statement. Record length is in words for unformatted data, and in characters for formatted data.
- In formatted output, be sure that the fields described by the format specifications are large enough to accommodate the output values. A number that is too large for its format specification produces a field of asterisks. For example, an attempt to print the number 1500.0 with the specification `F5.2` would produce `*****`.

## Programming for Easier Debugging

Programming techniques play an important role in the debugging process. Not only is a well-written program clearer and therefore easier to debug, but it is also less likely to contain errors in the first place. Following are some suggestions for making a program easier to debug.

Use a Modular Structure

Avoid "Tricks"

Avoid Nonstandard Usages

Use comments to document your program

### Use a Modular Structure

A modular program is one that is organized into blocks of statements, each of which performs a single logical function. A modular program is generally easier to understand and easier to debug, since you can debug each module separately. A modular program is structured so that execution flows from the top to the bottom, with a minimum of branching.

You can modularize a program by organizing it so that logically distinct tasks are performed by separate program units. Not only does this make the program clearer, but it can also reduce the size of the program. A subprogram is included only once in the program's area, regardless of the number of times it is called. Also, a subprogram that is sufficiently general can be shared by other programs that require the same task.

A disadvantage of subprograms is that the additional instructions that must be executed to link to the subprogram and to return to the calling program can increase a program's execution time. And too many subprograms can defeat the original purpose of making the program clearer and easier to debug.

The following guidelines can help you decide when to use subprograms:

- If a subprogram is already available for a particular task, use it. It's probably not worth the time it would take you to write it yourself. The FORTRAN library, for example, provides functions that perform such commonly-required operations as square roots, trig functions, random number generation, and so forth.
- If a task is done more than once in your program, write a subprogram to do the task.
- If a task is likely to be needed by other subprograms, write a subprogram to do the task.

You can modularize the statements within a program unit by using block IF structures. These structures can simplify a program by eliminating branches.

Normally, to execute a sequence of statements based on the outcome of an IF statement, you must branch to those statements using a GO TO statement. However, block IF structures enable you to conditionally execute groups of statements (called if blocks) without the need for GO TO statements.

A program written using block IF structures is easier to read because

- The flow of execution is from the top of the program to the bottom.
- The sequences of conditionally-executed statements are close to the IF statements.

Block IF structures generate the same object code as GO TO statements, and therefore do not affect the efficiency of a program.

The block IF statements are

```
IF (relational-expression) THEN
    ELSE IF (relational-expression) THEN
    ELSE
    END IF
```

Every block IF structure begins with an IF statement and ends with an END IF statement. A structure can optionally contain one or more ELSE IF statements and one ELSE statement.

The simplest block IF structure begins with with an IF statement, ends with an ENDIF statement, and contains no ELSE or ELSE IF statements:

```
IF (J .EQ. 1) THEN
    A = B + C
END IF
```

If J is equal to 1, the statement A = B + C is executed. Otherwise, execution continues with the statement following END IF.

You can form more complicated structures using ELSE and ELSE IF statements:

```
IF (NUM .GT. LIMIT) THEN
    NUM = 0
ELSE IF (NUM .GT. MANY) THEN
    NUM = NUM - 1
ELSE
    NUM = NUM + 1
END IF
```

The following example shows a program written in two ways. The first way uses relational IF statements, and the second way uses block IF statements.

Program with relational IF statements:

```
IF (I .GT. J) GO TO 30
IF (I .LT. J) GO TO 20
GO TO 40
20 A = A - 1.0
   B = B - 1.0
   GO TO 40
30 A = A + 1.0
   B = B + 1.0
40 CONTINUE
```

Program with block IF statements:

```
IF (I .LT. J) GO TO 20
    A = A - 1.0
    B = B - 1.0
ELSE IF (I .GT. J) THEN
    A = A + 1.0
    B = B + 1.0
END IF
```

The block if structure requires no branching, eliminates the need for statement labels, and requires fewer statements.

## Example of a Modular Program

The following example illustrates a modular program. The program is written so that each distinct operation is performed by a separate subroutine or function. Don't worry about the processing details. The purpose of this example is to show how a program can be organized into modules.

The example program (program SALES) reads monthly or weekly sales data and tabulates the data in chronological order and in order of decreasing sales. All input and output is interactive (through the terminal). The program assumes that the data is input in chronological order.

Program SALES performs the following distinct functions:

- Reads input data
- Sorts the data
- Prints a report

The sort portion of the program can be further subdivided into the following functions:

- Search the array to be sorted
- Switch the positions of two values in the array

The program is modularized into the following tasks:

- The main program reads input data and calls subroutine REPORT.
- Subroutine REPORT calls subroutine SORT and prints monthly or weekly figures in chronological order and decreasing order.
- Subroutine SORT locates the largest value in an unsorted array and calls subroutine SWITCH to interchange the largest value with the first value in the array.
- Subroutine SWITCH interchanges the values of the two input variables.

Although this program is too small to gain much efficiency as a result of modularization, it is much more readable than if it were written as a single program unit. Furthermore, subroutines SORT and SWITCH are generalized. That is, they could be used by any program requiring a sort.

The following main program reads the year and the interval (weekly or monthly).

```
PROGRAM SALES
CHARACTER IPER*7, YEAR*4
```

C Prompt user for input.

```
5 PRINT*, ' YEAR?'
```

```
READ '(A4)', YEAR
IF (YEAR .EQ. 'END') STOP
```

C Prompt user for input.

```
PRINT *, ' WEEKLY OR MONTHLY?'
READ '(A7)', IPER
```

C N is the number of sales periods. N=52 for weekly reports;

C N=12 for monthly reports.

```
IF (IPER .EQ. 'WEEKLY') THEN
  N = 52
ELSE
  N = 12
ENDIF
```

C Subroutine REPORT generates the reports.

```
CALL REPORT (YEAR,N)
```

C Branch to the beginning to generate another report or

C terminate.

```
GO TO 5
END
```

Subroutine REPORT reads sales figures and prints the requested report. The input values in array ISALES are copied to array NSORT. Array ISORT contains the number of each month or week of the year.

```

SUBROUTINE REPORT (Y, N)
  CHARACTER TYPE(2)*7, Y*4
  INTEGER ISALES(52), ISORT(52), NSORT(52), T
  DATA TYPE/' WEEKLY', 'MONTHLY'/

  C Determine if weekly or monthly sales period, then set index.
  C T=1 means weekly reports, T=2 means monthly reports.
  IF (N .EQ. 52) THEN
    T = 1
  ELSE
    T = 2
  ENDIF

  C Prompt user for input.
  PRINT *, ' ENTER SALES FIGURES'

  C Read sales figures.
  READ *, (ISALES(I),I=1,N)

  C Print table heads.
  PRINT 1000, TYPE(T), Y, TYPE(T)(1:5), TYPE(T)(1:5)

  1000 FORMAT (//A7,' SALES FIGURES FOR ',A4,//3X,'CHRONOLOGICAL',
    +      12X,' BEST TO WORST',/2(A5,9X,' SALES',6X))

  C Initialize arrays for SORT routine.
  DO 10 I=1,N
    ISORT(I) = I
    NSORT(I) = ISALES(I)
  10 CONTINUE

  C Sort arrays ISORT and NSORT into decreasing order.
  CALL SORT (N, ISORT, NSORT)

  C Print sales figures.
  DO 20 I=1,N
    PRINT 101 I,SALES(I),ISORT(I),NSORT(I)
  20 CONTINUE
  101 FORMAT (1X,2(I1,8X,I6,11X))

  RETURN
  END

```



Subroutine SORT sorts the two input arrays IA and IB into descending order. N is the number of values to be sorted. The sort is performed by finding the largest value and placing it in the first word of the array, finding the next largest value and placing it in the second word, and so forth.

```

SUBROUTINE SORT (N, IA, IB)
  DIMENSION IA(N), IB(N)

  DO 55 I=1,N-1
    LARGE = I

    DO 50 J=I+1,N
      IF (IB(J) .GT. IB(LARGE)) LARGE = J
50    CONTINUE

    CALL SWITCH (IA(I), IA(LARGE))
    CALL SWITCH (IB(I), IB(LARGE))
55  CONTINUE
  RETURN
  END

```

Subroutine SWITCH interchanges the values of I and J.

```

SUBROUTINE SWITCH (I, J)
  ITEMP = I
  I = J
  J = ITEMP
  RETURN
  END

```

Remember that modularity is achieved at a certain expense. The increased number of function and subroutine calls result in increased execution time. And if you are not careful, the numerous subprograms can result in less efficient use of virtual memory. (Efficient use of virtual memory is discussed in chapter 5, Using Virtual Memory.)

### Avoid "Tricks"

Avoid techniques that are obscure, especially when there is an easier way. For example, the following sequences show two ways of creating an identity matrix (a matrix with ones on the main diagonal and zeros elsewhere):

Method 1:

```

DO 10 I=1,N
  DO 10 J=1,N
    A(I,J) = (I/J) * (J/I)
10  CONTINUE

```

Method 2:

```

      DO 10 I=1,N
        DO 8 J=1,N
          A(I,J) = 0.0
      8   CONTINUE
        A(I,J) = 1.0
    10  CONTINUE

```

The first method is a little shorter, but it's somewhat more confusing. The "trick" is that the term  $(I/J)*(J/I)$  is one only when  $I=J$ ; the term is zero for all other values of  $I$  and  $J$ . The second sequence is longer, but more straightforward. (It zeros each row and sets the diagonal element to one.)

## Avoid Nonstandard Usages

Nonstandard usages make a program harder to understand for people who aren't familiar with the usage. They also may require rewriting if the program is to be run on another system. There is also the possibility that a particular nonstandard feature will be deleted in a future version of the language.

The FORTRAN Language Definition manual clearly indicates those features that are nonstandard. The manual also contains a summary of all the nonstandard features. In most cases, you can replace a nonstandard feature with a standard one.

You can request the FORTRAN compiler to diagnose all non-ANSI usages in a program by specifying the `STANDARDS_DIAGNOSTICS` parameter on the FORTRAN command. This command allows you to specify the level of severity at which you want the usages to be diagnosed. For example, the command

```
FORTRAN INPUT=PROGA STANDARDS_DIAGNOSTICS=WARNING
```

causes the compiler to issue warning messages for all nonstandard usages.

Following are some of the more commonly-used nonstandard features and the standard features by which they can be replaced. Refer to the FORTRAN Language Definition for a more comprehensive list.

<u>Nonstandard Feature</u>	<u>Equivalent Standard Feature</u>
boolean forms <code>L"..."</code> , <code>R"..."</code> , and <code>"..."</code>	CHARACTER data type
ENCODE and DECODE	Internal READ and WRITE

<u>Nonstandard Feature</u>	<u>Equivalent Standard Feature</u>
File declarations on the PROGRAM statement	OPEN statement
READMS and WRITMS	direct access READ and WRITE
BUFFER IN and BUFFER OUT	unformatted READ and WRITE
Intrinsic functions AND, OR, XOR, NEQV, EQV and COMPL	logical operators .AND., .OR., .NOT., .EQV., and .NEQV.

Following are some additional suggestions for improving the readability of a program.

- Use comments to document your program. Comments are especially desirable before each major step of a program (such as DO loops and IF blocks) and wherever the program does something that is not obvious.
- Indent IF blocks and the ranges of DO loops by at least two spaces.
- Use blanks to set off the components of a statement. For example,

```
IF(A.EQ.B.OR.A.EQ.C)GOTO25
```

should be written as

```
IF (A .EQ. B .OR. A .EQ. C) GO TO 25
```

- Use meaningful names for variables and arrays. For example, the statement

```
A = B*C
```

is not as meaningful as

```
SALARY = RATE*HOURS
```

- Arrange statement labels in numerical order, and use labels of different sizes for FORMAT statements and executable statements. For example, use 3-digit labels for executable statements and 5-digit labels for FORMAT statements.

## Summary of the Debugging Process

An important step in the development of a program is the detection and correction of errors in that program.

Before you attempt to execute your program, you should be sure all compilation errors have been corrected. When you compile the program, some suggested options are:

Omit the `OPTIMIZATION` parameter for fast compilation and slow execution.

Specify `LIST_OPTIONS=(S,A,R,M)` to generate a reference map.

If executing at a terminal, specify `LIST=file-name` to designate a file to receive compiler output.

Specify `BINARY_OBJECT=$NULL` so that an object program is not produced.

Before you begin execution of your program, enter the following commands:

```
SET_MESSAGE_MODE FULL
```

This command causes error messages to be issued in their complete form.

```
SET_PROGRAM_ATTRIBUTE PRESET=INDEFINITE
```

or

```
SET_PROGRAM_ATTRIBUTE_PRESET=INFINITY
```

This command initializes all variables and arrays to an indefinite or infinite value.

Some of the more common types of execution errors occur during arithmetic operations. The errors generally occur either when a computation generates an infinite value (exponent or arithmetic overflow) or when an infinite or indefinite value is used in a computation.

Error conditions involving exponent overflow, arithmetic overflow, and infinite or indefinite operands are usually caused by programming errors such as use of undefined variables, incorrect input data, or division by zero. These kinds of errors can be difficult to locate because, in many cases, the statements that generate the incorrect values are widely separated from the statements that actually generate the error condition.

Following are some common mistakes that can lead to runtime errors:

- Undefined or improperly initialized variables
- Misspelled or mistyped variable names
- Subscripts that exceed declared dimension bounds
- Improper use of mixed mode arithmetic
- Incorrect or invalid input data
- Mismatch between dummy arguments in subprogram argument list and actual arguments in subprogram reference
- Accumulated roundoff error
- Incorrect logic.

The following guidelines can help eliminate input/output errors:

- Avoid mixing I/O types on the same file.
- Always specify the END= specifier on READ statements.
- In formatted I/O, be sure that the list items are consistent with the associated format specifications.
- In direct access I/O, be sure you specify the correct record length.

By using clear and careful programming techniques, you can minimize the number of errors that occur in a program and produce a program that is easier to debug. Some programming guidelines are as follows:

- Use block IF structures instead of logical or arithmetic IF statements.
- Organize your program into functions and subroutines that perform logically distinct tasks.
- Avoid programming "tricks".
- Avoid nonstandard usages.
- Use meaningful variable names.
- Indent DO loops and block IF structures.
- Use spaces to improve readability in FORTRAN statements.
- Use comments liberally.

## Debugging Aids

NOS/VE and NOS/VE FORTRAN provide tools to help you debug your programs. These tools include:

- The FORTRAN reference map
- The FORTRAN C\$ Directives
- The HELP utility
- The Debug utility

## Using the Reference Map

The reference map is a listing produced by the compiler, which provides information about all the symbolic names used in a program. This information can be very useful in helping you detect errors in the program. When you are debugging a program, you should always request a reference for each compilation.

A separate map is produced for each program unit in the program. The map is divided into the following sections:

- Symbolic constants map
- Namelist map
- Variables map
- Common and equivalence map
- Statement labels map
- DO loops map
- Entry points map
- Procedures map
- Input/Output units map
- Unclassified names map

Each section lists information about a particular type of program entity. Although each of these sections can be useful, probably the most useful sections in terms detecting common errors are the Variables section and the Statement Labels section.

## Generating a Reference Map

Generation of the reference map is controlled by the `LIST_OPTIONS` parameter on the FORTRAN command. This parameter has several options, each of which controls a different aspect of the compiler output listing. You can request a complete reference map by specifying the `LIST_OPTIONS` options shown in the following FORTRAN command example:

```
FORTRAN INPUT=SRCE LIST_OPTIONS=(S,A,R,M) LIST=MAPFIL
```

In this example, the source program is on file `SRCE`. The source listing and reference map are written to file `MAPFIL`.

If you are executing interactively, do not forget to specify the `LIST` parameter on the FORTRAN command. This is the file that receives all compiler output. If you omit this parameter, the source listing and reference map will be discarded.

## What the Variables Map Tells You

The variables map contains six columns of information, identified by the following titles:

```
NAME SECTION+OFFSET SIZE PROPERTIES TYPE REFERENCES
```

Under `NAME`, the variables defined and referenced in the program unit are listed in alphabetical order.

The `SECTION+OFFSET` column gives the relative hexadecimal address of each variable. This information is useful mainly for locating variables in a memory dump (a practice that is recommended only as a last resort).

The `SIZE` column gives the number of elements in each array. For variables, the corresponding entry in this column is blank.

The `PROPERTIES` column contains symbols that give useful debugging information. In particular, look for the following symbols in this column:

- UND** If this symbol appears, the variable or array was not defined anywhere in the program unit. An undefined variable is one that is referenced but has not been stored into. Using undefined variables is a common source of error. This generally means either that you misspelled a variable name or that you forgot to initialize the variable.
- \*S\*** This is the "stray name" flag. A stray name is a name that appears only once in the entire program unit. It generally means either that the variable name is misspelled or that the variable is unneeded and can be eliminated.

The TYPE column gives the data type (REAL, INTEGER, CHARACTER, COMPLEX, BOOLEAN, DOUBLE PRECISION) of each variable.

The REFERENCES column gives the number of each source line where the variable is used (either defined or referenced). The line numbers may be suffixed by a symbol that describes how the variable was used. For example, the symbol /M means that the variable was used in a statement that modified its contents; the symbol /S means that the variable appeared in a subscript expression. (A legend at the bottom of each page of the reference map explains all of the symbols that can appear in this column.)

## What the Statement Labels Map Tells You

The statement labels map lists information about all statement labels used in the program. The map contains five columns of information, identified by the following titles:

LABEL SECTION+OFFSET PROPERTIES DEF REFERENCES

Under the LABEL title the labels are listed in numerical order.

The SECTION+OFFSET column gives the relative hexadecimal address of the labels. (Useful mainly for debugging the object listing). If the label is not defined anywhere in the program unit, the symbol \*UNDEFINED\* appears. If the label is defined but not referenced, the symbol \*NO REFERENCES\* appears.

The PROPERTIES column contains a symbol that indicates how each label was used:

blank	Used as the label of an executable statement.
FORMAT	Used as the label of a FORMAT statement.
DO-TERM	Used in a DO statement.
NON-EX	Used in a nonexecutable statement.

The DEF column contains the number of the source line where the label is defined. If the label is not defined (that is, if it does not appear in the label field of a statement), the symbol UNDEF appears. An undefined label is an error. For example, if the statement

```
GO TO 25
```

appears in a program, but the label 25 does not appear in the label field of an executable statement in the same program unit, the label is flagged as undefined.



The REFERENCES column gives the line numbers of all source statements that reference the label. If a label is not referenced anywhere in the program unit, it can be safely removed.

### Example of a Variables Map

To illustrate a reference map, the following program is compiled with the parameter LIST\_OPTIONS=(S,A,R,M) specified on the FORTRAN command.

```

PROGRAM TRIANGL
OPEN (UNIT=1, file='SIDES')
DO 10 I=1,100
  READ (UNIT=1, FMT=*, END=999, IOSTAT=IOS) S1, S2, S3
  S = (S1 + S2 + S3)/2.0
  ASQ = S*(S - S1)*(S - S2)*(S - S3)
  A = SQRT(ASQ)
  PRINT 100, S1, S2, S3, AREA
100 FORMAT (' SIDES= ', 3F6.2, /' AREA= ', F6.2, /)
10 CONTINUE
END

```

The following variables map is produced for this program:

--VARIABLES--

-NAME---SECTION+OFFSET---SIZE-PROPERTIES---TYPE-----REFERENCES

A	\$STATIC+72		*S*	REAL	8/M
AREA	\$STATIC+80		UND/*S*	REAL	9
ASQ	\$STATIC+64			REAL	7/M 8/P
S	\$STATIC+56			REAL	6/M 7 7 7 7
S1	\$STATIC+24			REAL	4/M 7 9
S2	\$STATIC+32			REAL	4/M 6 7 9
S3	\$STATIC+40			REAL	4/M 6 7 9
SL	\$STATIC+48		UND/*S*	REAL	6

By examining this map, we can see that the variables A, AREA, and SL have some unusual properties associated with them.

A has the property \*S\*. This means that A is a stray name (it is used only once in the source program).

AREA and SL have the property UND/\*S\*. This means that AREA and SL are stray names and are undefined (they are not stored into anywhere in the program).

We can use the REFERENCES part of the map to help us find the questionable variables in the source listing. The REFERENCES column gives the number of each source line where the variables appear.

Variables SL, A, and AREA are referenced in lines 6, 8, and 9, respectively:

```

line 6:  S = (SL + S2 + S3)/2.0 <----- variable SL is referenced
line 8:  A = SQRT(ASQ) <----- variable A is referenced
line 9:  PRINT 100, S1, S2, S3, AREA <---- variable AREA is
                                             referenced

```

Examining these lines reveals two errors: In line 6, SL should be changed to S1, and in line 9, AREA should be changed to A.

## Using the HELP Command to Obtain Error Descriptions

The HELP command provides online descriptions of execution-time errors. These descriptions are contained in the online manual named MESSAGES. The descriptions include an explanation of the error, some possible causes, and where to go for further information.

If you are using the system interactively and a runtime error occurs, your program terminates and the system displays a message such as:

```
--ERROR--Initial '(' missing from format spec.
```

If you want more information about the error message, simply type:

```
HELP
```

The system then takes you to the description of the error message in the MESSAGES online manual. (In the case where several error messages are issued, you are taken to the description of the most recent one.) (You can also use the command EXPLAIN MESSAGE, abbreviated EXPM. The HELP and EXPLAIN MESSAGE commands are equivalent.)

For example, if your program terminates with the message

```
--ERROR--Initial '(' missing from format spec.
```

and you then type

```
HELP
```

you are taken to the following description:

`--ERROR--Initial '(' missing from format spec.`

```
Condition: 585069            Product Identifier: FL
Condition Identifier: fl$open_paren_missing
```

**Description:**

A format specification must be bound by parentheses.

**User Action:**

Ensure that the number of left parentheses in the specification matches the number of right parentheses.

**Further Information:**

A selection from the following menu takes you to another online manual for further information. You can return to this screen by typing `REVERT` from the other online manual.

a. Description of the `FORMAT` statement.

Choose a topic, press `RETURN` for next message, or type `QUIT` to leave manual.

The Further Information portion of the description gives you the option of requesting further information about the error message. This option provides a direct link to the relevant online manual. Thus, in the preceding example, if you type the letter `a` (and then press `RETURN`), you are taken to the description of the `FORMAT` statement in the online FORTRAN Language Definition manual.

The preceding method of looking up an error message is useful if you are executing interactively and your program terminates with an error message. You can go directly to the description of the error message simply by typing `HELP` (or `EXPLAIN_MESSAGE` or `EXPM`). But this method works only if the error you want to read about is the most recent one to occur in the same terminal session.

If the error of interest occurred in another terminal session, or if other errors have intervened in the same terminal session, you must use the `EXPLAIN` command to enter the online `MESSAGES` manual and go to the description of the error. But in order to look up a particular error in the online `MESSAGES` manual, you must know the condition code associated with that error.

Every error message is identified by a unique six-digit number called a condition code. You can direct the system to display the associated condition code whenever it issues an error message by setting message mode to FULL. The command to set message mode to FULL is:

```
SET_MESSAGE_MODE FULL
```

In the absence of this command, message mode is automatically set to BRIEF, and condition codes are not displayed with error messages.

After you enter a SET\_MESSAGE\_MODE FULL command, FULL mode remains in effect either until the end of the terminal session or until you enter a SET\_MESSAGE\_MODE BRIEF command.

An example of an error message issued in FULL mode is:

```
585069--ERROR--Initial '(' missing from format spec.
```

You can enter the MESSAGES online manual and go directly to the description of the error by typing

```
EXPLAIN M=MESSAGES S='585069'
```

## Using the Compiler Control (C\$) Directives

Compiler control directives are statements that you can insert in your source program to control various aspects of compilation. Two types of directives can help you debug your program. These are the conditional compilation directives and the listing control directives. Refer to the FORTRAN Language Definition manual for detailed descriptions of these and other C\$ directives.

### Conditional Compilation Directives

The conditional compilation directives enable you to selectively compile portions of a program, while causing the compiler to disregard other parts.

Using these directives, you can insert debugging statements in a program, and then tell the compiler either to compile or to not compile those statements, depending on the result of a conditional test. Thus, you do not need to physically remove the statements from the program and then insert them again when you want the debug printout.

The conditional compilation directives are:

```
C$ IF (logical-expression)
```

```
C$ ENDIF
```

If the logical-expression is true, the statements between C\$ IF and C\$ ENDIF are compiled as usual. If the logical-expression is false, the statements are not compiled. An optional C\$ ELSE directive allows you to specify an alternate block of statements.

The following example illustrates the C\$ IF and C\$ ENDIF directives:

```

PARAMETER (N=0)
.
.
.
C$ IF (N .EQ. 1)
PRINT 99 (X(I),I=1,100)
C$ ENDIF
.
.
.

```

In this example, the PRINT statement is disregarded because the value of the expression in the C\$ IF directive is false. If the programmer wanted to compile and execute the PRINT statement, he or she would simply change the PARAMETER statement to set N to 1, so that the expression in the C\$ IF statement would have the value true.

## Listing Control Directives

The FORTRAN compiler, through the LIST\_OPTIONS (LO) parameter, allows you select various output listing options. These options include a complete listing of the source program and a reference map that provides detailed information about all the symbolic names used in the program. However, this output listing can become quite lengthy. The listing control directives enable you to reduce the volume of output by suppressing any or all of the LO options for selected parts of a program.

The listing control directive has the form

```
C$ LIST (p=c)
```

where p is one of the symbols S, O, R, A, M, or ALL (indicating which listing options is to be suppressed), and c is a constant (or symbolic constant defined by a PARAMETER statement) having the value 0 or 1.

When the compiler encounters a C\$ LIST directive of the form C\$ LIST(p=1) the list options specified by p are turned off. Those options remain off until either the end of the program unit is encountered or the directive C\$ LIST(p=0) is encountered. The list options are then turned back on.

C\$ LIST directives can be useful for debugging modular programs, because you can use the directives to suppress source listing options for modules that you are not currently debugging. (Although you can insert the C\$ LIST directives anywhere in your source program, you will typically insert them at the beginning of program units.)

The following simple example illustrates the C\$ LIST directive.

```
PROGRAM MAIN
C$ LIST (ALL=0)
.
.
.
END

SUBROUTINE SUB1
.
.
.
RETURN
END
```

Assume that this program resides on file PROGA and is compiled with the command

```
FORTRAN INPUT=PROGA LIST_OPTIONS=(S,R) LIST=LFILE
```

Both the main program and subroutine are compiled as usual. But only subroutine SUB1 appears in the source listing (S option) and reference map (R option). Note that because the C\$ LIST directive affects only the program unit in which it appears, no C\$LIST directive is needed at the beginning of the subroutine to turn the list options option back on.

## Using Debug

Debug is a utility program that helps you debug a program during execution. Through Debug, you can stop execution at selected points, display the values of selected variables and arrays, and resume execution. The displayed values are formatted according to a default format that is consistent with the formats of FORTRAN values. You specify the symbolic names of the locations you want to display; no knowledge of machine addresses is required.

A primary advantage of Debug is that it is easy to use. It requires no modification of the source code, and no knowledge of assembly language. Further, it eliminates the need for such conventional debugging techniques as interpreting memory dumps, inserting PRINT statements within a program, and using a load map.

Other Debug features enable you to

- Change the values of program variables while execution is suspended.
- Display a subprogram traceback list, beginning with the current subprogram and proceeding back through the sequence of called subprograms until the main program is reached.
- Create a file of Debug commands that Debug will execute only if an execution error occurs in your program. If an execution error does not occur, the program runs as though Debug were not being used.

Using Debug involves the following sequence of steps:

1. Compile your program for use with Debug.
2. Turn on debug mode.
3. Begin the Debug session.
4. Enter Debug commands to debug your program.
5. End the Debug session.
6. Make corrections to your source program, recompile, and, if necessary, conduct additional Debug sessions.

Later, we will take a closer look at these steps. First, we find out exactly what a Debug session is.

A Debug session is the sequence of interactions that takes place between you and Debug after you begin execution of a program in debug mode. Each time you enter a command, debug processes that command, displays any output or informative messages produced by the command, and waits for you to enter another command.

The Debug session begins when you enter a name call or EXECUTE\_TASK command to begin execution of a program while in debug mode. Normally, this begins execution of your program. However, in debug mode, control immediately transfers to Debug, which displays the following prompt:

DB/

and waits for you to enter a Debug command. After you enter a command, Debug processes that command and issues another DB/ prompt. This sequence of interactions continues until you end the Debug session.

A typical Debug session proceeds according to the following sequence of events:

1. You turn on debug mode. You can do this with either a SET\_PROGRAM\_ATTRIBUTE command or an EXECUTE\_TASK command.
2. You specify a name call or EXECUTE\_TASK command. This begins the Debug session. Debug immediately gets control and waits for you to enter a command.
3. You make provisions for getting control during execution of your program. This involves using the SET\_BREAK command to set breaks at the source lines where you want execution to stop.
4. You begin execution of your program by entering a RUN command. The program executes until either a break is encountered or an execution error occurs. Then Debug gets control and prompts you for a command.
5. While execution is suspended, you enter commands to display or change the values of program variables and arrays.
6. When you are ready to resume execution, enter a RUN command. The program continues executing until it encounters another break or execution error. You can repeat steps 5 and 6 until the program terminates or you are ready to end the Debug session.
7. To end the Debug session, type QUIT. This returns control to the operating system.



Following is a simple example of a Debug session:

```

/execute_task (file=lgo, debug_mode=on) <-- Begin the Debug session.
DB/ set_break name=b1 line=25 <----- Set a break at line 25 of
                                     the program.
DB/ run <----- Begin execution of the
                                     program.
DEBUG: BREAK B1 at L=25 <----- The program runs until
                                     the break is encountered.
DB/ display_program_value name=aval <----- Display the value of
                                     variable AVAL.
AVAL = 3.14159 <----- Debug displays the value
                                     of AVAL.
DB/ run <----- Resume execution.
DEBUG: program terminated <----- The program terminates.
DB/ quit <----- End the Debug session.

```

## Preparing for a Debug Session

As previously noted, using Debug requires no changes to your source program. However, there are two FORTRAN command parameters you should know about if you intend to use Debug. These are the DEBUG and OPTIMIZATION parameters.

The important fact to remember about these parameters is that they default to produce the information needed to use Debug.

The DEBUG parameter on the FORTRAN command controls the generation of symbol and line number tables during compilation. The presence of these tables in the compiled program enables you to reference locations (variables, arrays, and source lines) by their symbolic name rather than by machine address. If these tables are not present, you can still use Debug, but you must specify all program locations by machine addresses. For instance, in order to display the variable X, you would have to specify the machine address of X.

Obviously, using machine addresses is much more difficult than using symbolic names. The only reason you might want to debug a program without the use of symbolic names is in the case of an existing object program that would be too costly to recompile. However, in most cases, it would probably be worthwhile to recompile the program and request the compiler to generate the symbol and line number tables.

The `DEBUG` parameter has the following options:

`DEBUG=NT` Requests the compiler to omit the tables.

`DEBUG=NONE` Requests the compiler to generate the tables.

To cause the compiler to generate the symbol and line number tables, you can either OMIT the `DEBUG` parameter (symbol and line number table generation is the default option) or specify `DEBUG=NONE`. This is the reverse of the way you might think it would work. Specifying `DEBUG=NT` causes the compiler to omit the tables.

After your program is debugged and you are through with Debug, you can recompile with `DEBUG=NT` specified on the FORTRAN command, so that tables are not generated and the program has the smallest possible size.

The `OPTIMIZATION` parameter controls the level of optimization performed by the compiler. Optimized object code executes faster than unoptimized code, but requires more compilation time. The `OPTIMIZATION=DEBUG` option selects minimum optimization (same as `OPTIMIZATION=LOW`) and also generates object code that is modified for more efficient use with Debug.

You can still use Debug with `OPTIMIZATION=HIGH`, but some lines may be removed from the program during optimization and some variables may not be available.

Since `DEBUG` is the default option for the `OPTIMIZATION` parameter, you can omit the `OPTIMIZATION` parameter when compiling for use with Debug. Then, when the program is completely debugged, you can recompile with `OPTIMIZATION=HIGH` for a more efficient object program.

In order to execute a program under Debug control, you must first request a mode of execution called debug mode. You can turn on debug mode in two ways.

The first way is to specify the `DEBUG_MODE=ON` option on a `SET_PROGRAM_ATTRIBUTES` command. All subsequent program executions will take place under Debug control, until you turn debug mode off with the `DEBUG_MODE=OFF` option. For example, the following sequence turns on debug mode, executes the programs on files `LGO1` and `LGO2`, and then turns debug mode off. Both executions take place under Debug control (assuming no intervening `DEBUG_MODE=OFF` specification.)

```
SET_PROGRAM_ATTRIBUTES DEBUG_MODE=ON
LGO1
.
.
.
LGO2
SET_PROGRAM_ATTRIBUTES DEBUG_MODE=OFF
```

The second way to turn on debug mode is to use an EXECUTE\_TASK command to begin execution. EXECUTE\_TASK is a NOS/VE command that begins execution of a program while enabling you to select various execution-time options. You can use EXECUTE\_TASK to turn on debug mode by specifying the DEBUG\_MODE=ON parameter. This turns on debug mode only for a particular execution of a program. For example,

```
EXECUTE_TASK (FILE=LGO, DEBUG_MODE=ON)
```

turns on debug mode and begins execution of file LGO.

Using the SET\_PROGRAM\_ATTRIBUTES command to turn on debug mode is useful if you will be conducting several Debug sessions within a single terminal session. If you want to use Debug only for a single execution of a program, you can use the EXECUTE\_TASK command.

If you have turned on debug mode through a SET\_PROGRAM\_ATTRIBUTES command, you can subsequently turn debug mode off for a particular execution of a program by specifying DEBUG\_MODE=OFF on the EXECUTE\_TASK command for that program.

To begin a Debug session, you must be sure that you have compiled for use with debug and turned on debug mode. You then begin the Debug session simply by beginning execution of your program in a normal manner; that is, by specifying a name call command (such as LGO) or an EXECUTE\_TASK command.

When you specify a name call or EXECUTE\_TASK command, and debug mode is on, your program does not begin executing. Instead, control transfers immediately to Debug. Debug then displays a message indicating that it has control, followed by a prompt, and waits for you to enter a Debug command. For example, the following sequence turns on debug mode and begins a Debug session.

```
set_program_attributes debug_mode=on
lgo
DEBUG
DB/
```

The string DB/ is the Debug prompt for user input.

Having begun the Debug session you are now ready to enter some Debug commands.

## Entering Debug Commands

During a Debug session, you enter Debug commands in response to prompts. Debug processes each command, displays any output and informative messages produced by the command, and issues another prompt. The Debug commands enable you to provide for receiving control during program execution, to display or change the values of variables and arrays within the program, to resume execution, to end the Debug session, and to perform a variety of other tasks.

Although Debug offers a large number of commands, only the most useful are discussed here. These commands enable you to:

Suspend Program Execution (SET\_BREAK and SET\_STEP\_MODE commands)

Begin or Resume Program Execution (RUN command)

Display Program Values (DISPLAY\_PROGRAM\_VALUE command)

Change Program Values (CHANGE\_PROGRAM\_VALUE command)

End the Debug Session (QUIT command)

You can also create a file of Debug commands that are executed automatically if an error occurs while your program is executing. Other commands described in this chapter enable you to display a traceback list and to display status information about the Debug session.

## Suspending Program Execution

At the beginning of a Debug session you will typically provide for receiving control at various points in the program, so that you can display and alter program values. You can cause Debug to suspend execution and give control to you in the following ways:

- When a specific line or statement is reached during execution. This requires you to set one or more breaks in your program. When the executing program encounters a break, execution suspends and you get control.
- When an error occurs during execution. This requires no action on your part. If you simply turn on debug mode and begin execution of your program, you will get control if a runtime error occurs.
- Immediately before execution of each source line in the program. This is known as executing in step mode. Step mode enables you to step through your program one line at a time.

We discuss breaks first. A break is a device that enables you to suspend execution of your program and receive control. Debug provides several different types of breaks. Each type of break gives control to you when a specific condition occurs during execution of your program.

When a break is encountered during program execution, the following sequence of events occurs:

1. Execution of the program immediately suspends.
2. Control passes to Debug.
3. Debug displays a message indicating the type of break that occurred and the location (module and line number) of the break. An example of such a message is:

```
-- DEBUG: break B1, execution at M=PROGA L=4
```

If the break was caused by an execution error, Debug displays a message describing the error.

4. Debug displays a DB/ prompt and gives control to you. You can then enter Debug commands.

The most useful break is the **LINE** break. This break gives control to you when a specific line is reached during execution. You set a break by entering a **SET\_BREAK** command. The command to set a **LINE** break has the form:

```
SET_BREAK BREAK=name LINE=line-number
```

This command sets a break at the line having the specified line number. (Line numbers are listed on the program source listing. The first line of each program unit is numbered 1, the second is numbered 2, and so forth.)

The **BREAK** parameter specifies a name of your choosing that you assign to the break. You can use this name to reference the break in other Debug commands, as described later in this chapter.

Following is an example of the **SET\_BREAK** command:

```
SET_BREAK BREAK=B1 LINE=50
```

This command sets a break at line 50 of the source program. The name **B1** is assigned to the break. When line 50 is reached during execution, execution will temporarily stop and you can enter Debug commands.

Suppose you are debugging a program that contains several subprograms. In this case, the `SET_BREAK` command described above applies to the program unit that was executing when Debug gained control. (At the beginning of a Debug session, Debug commands apply to the main program.) But suppose you want to set a break in a different program unit. For example, at the beginning of a Debug session, you might want to set a break in a subroutine or function within the program.

The optional `MODULE` parameter is designed for programs with multiple program units. If the line where you want to set the break is in a program unit other than the one where execution is currently suspended, the `MODULE` parameter enables you to specify the desired program unit. For example, the command

```
SET_BREAK BREAK=B1 LINE=50 MODULE=SUBA
```

sets a break at line 50 of program unit SUBA, regardless of where execution is currently suspended.

In addition to setting `LINE` breaks in your program, you can also get control during a Debug session simply by beginning execution of the program and allowing it to execute until an error occurs. Debug then gets control, displays a message describing the error, and gives control to you. This method of getting control during a Debug session is a result of the default error termination breaks provided by Debug. Examples of error termination breaks are the `DIVIDE_FAULT` break and the `EXPONENT_OVERFLOW` break. You do not need to explicitly set these breaks because they are default breaks that are automatically set for every Debug session.

In addition to the `LINE` break and error termination breaks, there are several other breaks you can set during a Debug session. Many of these breaks involve suspending execution at certain machine-level instructions, and are generally not useful to FORTRAN programmers.

A suggested procedure for debugging a program is to execute the program in debug mode without setting any `LINE` breaks, and to allow the program to terminate. If the program terminates with errors, you can run additional Debug sessions using knowledge gained from the first session to help you decide where to set `LINE` breaks.

A third method of getting control during a Debug session is to set step mode. Step mode is a mode of execution in which execution is suspended immediately before the execution of each program statement. Step mode allows you to step through your executing program one line at a time, displaying or changing program values at each step.

The command to set step mode in a program unit has the form:

```
SET_STEP_MODE MODE=ON
```

This command causes execution to suspend immediately before execution of each executable statement in the program. Execution remains suspended until you enter the command to resume execution.

The `SET_STEP_MODE` command applies only to the program unit where execution is suspended when you enter the command. To set step mode in a subprogram, you must set a line break at the first line of the subprogram, then set step mode when execution is suspended at that break.

Suppose you are running a Debug session for a program named `MYPROG`. The following command, entered at the beginning of a Debug session, sets step mode:

```
SET_STEP_MODE MODE=ON
```

After you begin execution of the program, you will get control as shown:

```
PROGRAM MYPROG
DIMENSION A(100)
OPEN (UNIT=2, FILE='XXX') <-- Execution stops here.
READ (2, 100) A <----- Execution stops here.
100 FORMAT (50E12.4)
CALL SUB (A) <----- Execution stops here.
END <----- Execution stops here.
```

Nonexecutable statements are not affected by the `SET_STEP_MODE` command, nor will execution suspend on any statements in subroutine `SUB`.

Suppose you have set step mode for a particular program unit, execution is suspended part way through the program unit, and you want to execute the rest of the program unit without any more suspensions of execution. You can turn step mode off by specifying the following command:

```
SET_STEP_MODE MODE=OFF
```

## Beginning or Resuming Program Execution

You use the `RUN` command to begin execution of your program after beginning a Debug session, and to resume execution while in step mode or after the occurrence of a break. This command has the form

```
RUN
```

When you enter a `RUN` command, execution begins at the point where it was suspended. Execution continues until either another break occurs or the program runs to completion.

If you enter a RUN command after your program has run to completion (remember that you automatically get control when the program terminates) the Debug session automatically ends and control returns to the operating system.

The following Debug session shows how the RUN command works.

```
execute_task file=lgo debug_mode=on <-- Begin the Debug session.

DB/set_step_mode mode=on <----- Turn on step mode.

DB/run <----- Begin execution of the
                    program.

--DEBUG: step at M=SAMPLE L=2 <----- Execution stops before
                    execution of line 2
                    (first executable line).

DB/run <----- Resume execution of the
                    program.

--DEBUG step at M=SAMPLE L=3 <----- Execution stops before
                    execution of line 3.

DB/run <----- Resume execution of the
                    program.

.
.
.
```

### Displaying Program Values

After Debug has suspended execution of your program, you can display the values of variables and arrays with the DISPLAY\_PROGRAM\_VALUE command. This command has the form:

```
DISPLAY_PROGRAM_VALUE NAME=var
```

where var is the name of the variable or array element whose value is to be displayed. For example, the following commands display the values of the variable YVAL and element 99 of array ARR, respectively:

```
DISPLAY_PROGRAM_VALUE NAME=YVAL
```

```
DISPLAY_PROGRAM_VALUE NAME=ARR(99)
```

If you specify an array name without subscripts, the ENTIRE ARRAY is displayed.



The formats of values displayed by a `DISPLAY_PROGRAM_VALUE` command are consistent with the forms of FORTRAN constants. For example, if two variables A and B have the following values:

A: 'HELLO' (A is declared type character)

B: .001385

Then A and B are displayed as follows:

```
DISPLAY_PROGRAM_VALUE NAME=A
```

```
A = HELLO
```

```
DISPLAY_PROGRAM_VALUE NAME=B
```

```
B = .001385
```

Now suppose you are debugging a program that contains several program units. If you use the form of the `DISPLAY_PROGRAM_VALUE` command shown in the preceding examples, the values displayed are the ones in the program unit that was executing when Debug gained control.

To display values in a program unit other than the one where execution is currently suspended, use the following command:

```
DISPLAY_PROGRAM_VALUE NAME=var MODULE=prog
```

where `prog` is the name of the program unit containing the specified variable or array. By using this form of the `DISPLAY_PROGRAM_VALUE` command, you can display values in any program unit. For example:

```
PROGRAM MYPROG
A = 1.2 <----- If you want to display the value of the
.                variable A ...
.
.
CALL SUB
.
.
SUBROUTINE SUB
X = ... <----- ... and execution is suspended here, enter the
.                command
.
.                DISPLAY_PROGRAM_VALUE NAME=A..
.                ..MODULE=MYPROG
END
```

## Changing Program Values

We have seen how you can use Debug to suspend execution of your program and to display the values of variables and arrays within the program. Now, suppose execution of your program is suspended, and you have displayed the value of a variable, and that value is incorrect. Debug provides a command that enables you to replace that value with a new value. Then, when you resume execution (using the RUN command), the new value is used in subsequent computations.

The command to change the value of a variable or array element is `CHANGE_PROGRAM_VALUE`. This command has the form:

```
CHANGE_PROGRAM_VALUE NAME=var VALUE=val
```

where `var` is the variable or array element to be changed, and `val` is the new value to be assigned to `var`.

For example, suppose execution is suspended at line 4 of the following program:

```

1      PROGRAM SAMPLE
2      A = 5.0
3      B = 3.0
4      C = A*B <----- Execution is suspended here.
5      PRINT*, A, B, C
6      END

```

You can supply a new value for the variable A by entering the command

```
CHANGE_PROGRAM_VALUE NAME=A VALUE=10.0
```

This command places the value 10.0 in the variable A.

When you resume execution, the new value is used in the expression `A*B` and the `PRINT` statement prints

```
10.0 3.0 30.0
```

## Ending the Debug Session

You can end a Debug session at any time by typing the command

```
QUIT
```

When you enter a `QUIT` command, Debug displays the message

```
-- DEBUG QUIT terminated task
```

and returns control to the operating system.

Note that changes made during a Debug session are lost when the session is ended. All variables assume their original values, breaks are removed, and the program is the same as when you compiled it. You can run additional sessions if you want to continue debugging your program.

## Displaying a List of Breaks

You can display a list of all the breaks currently set in a program by specifying a SETBREAK command. This command has the form

### DISPLAY\_BREAK

The list includes the name of the break (as specified in the SET\_BREAK command that defined the break) and the location of the break (program unit name and source line number).

This command is particularly useful in longer Debug sessions where you are setting and removing numerous breaks, because it provides a way for you to keep track of which breaks are in effect at any given time.

Following is a typical example of output generated by a DISPLAY\_BREAK command:

```
--Break BRK1
  event(s)= execution
  location: M=MYPROG L=3

--BREAK BRK2
  event(s)= execution
  location: M=SUBC L=6
```

This output indicates that two breaks are set: A break named BRK1 is set at line 3 of program MYPROG, and a break named BRK2 is set at line 6 of subroutine SUBC.

## Removing Breaks from a Program

You can remove breaks during a Debug session by entering a DELETE BREAK command. This command has the forms

```
DELETE_BREAK BREAK=(name1, . . . , namen)
```

```
DELETE_BREAK ALL
```

The first form deletes the breaks having the specified names. (name<sub>i</sub> is the name you assigned to the break in the SET\_BREAK command.) The second form deletes all of the breaks currently set in the program.

For example, the following command deletes breaks BR1, BR2, and BR4:

```
DELETE_BREAK BREAK=(BR1, BR2, BR4)
```

The `DELETE_BREAK` command is especially useful in longer Debug sessions. By removing breaks that are no longer needed, you can speed up the session. A simple example is as follows:

```
line 10      DO 5 I=1,1000
      .      .
      .      .
      .      .
line 20      5  CONTINUE
```

The following command sets a break at line 20 of the preceding DO loop:

```
SET_BREAK BREAK=LOOP5 LINE=20
```

Execution would suspend on each pass through the loop, a total of 1000 times. But suppose you wanted to check the status of the loop only on the first few passes through the loop. You could then use the command

```
DELETE_BREAK NAME=LOOP5
```

to remove the break at statement 5 after you were through checking the loop. Then, when you resumed execution, the loop would complete with no suspension.

## Displaying a Subprogram Traceback List

You can display a subprogram traceback list by using the `DISPLAY_CALL` command. This command has the form

```
DISPLAY_CALL
```

The traceback begins with the routine that was executing when Debug gained control, and proceeds through the sequence of called routines until the main program is reached. For each routine in the traceback, the `DISPLAY_CALL` command displays the routine name and the source line number from which the routine was called.

For example, consider the following sequence of subroutines:

```

line 1      PROGRAM MAIN
.           .
.           .
.           .
line 8      CALL SUB1

line 1      SUBROUTINE SUB1
.           .
.           .
.           .
line 4      Z = F(R,S)

line 1      FUNCTION F(X,Y)
.           .
.           .
.           .

```

Assume that the program is run under Debug control and that execution is suspended in function F. The command `DISPLAY_CALL` would produce the following traceback:

```

--Traceback from procedure F module F at line 3
--Called from procedure SUB1 module SUB1 at line 4 byte offset 12
--Called from procedure MAIN module MAIN at line 12 byte offset 12

```

The most pertinent items of information are the module names (in Debug terminology, a program unit is called a module) and the line numbers.

## Displaying Information About the Debug Session

Debug provides a command that displays information about the current debug session. This command has the form

```
DISPLAY_DEBUGGING_ENVIRONMENT
```

This command displays a paragraph of information about the status of the Debug session. For most programmers, the most useful entry in this paragraph tells you where execution is currently suspended. A typical example is as follows:

```
.  
. Execution is currently stopped at B 030 0000006C which,  
in higher symbolic terms, is M=MYPROG L=5.  
. .
```

This output indicates that execution has stopped at line 5 of program MYPROG. Generally, you can ignore the hexadecimal form of the address.

## Automatic Execution of Debug Commands

Debug provides a feature through which you can create a special file, called an `ABORT_FILE`, that contains Debug commands. If your program terminates because of an execution error, Debug gets control and automatically executes the commands in the file.

The `ABORT_FILE` feature is especially useful for maintaining programs that are in the working stage. It allows you to specify a set of commands that is executed automatically when an error occurs. This ability to perform a specific set of debugging operations at the time of an error can eliminate the need to reproduce the error in order to debug the program.

In order to use the `ABORT_FILE` feature, debug mode must be off. The first step is to create a text file containing the commands you want to be executed. Simply use a text editor to enter the commands into a file in the same form as you would enter them in a Debug session.

The next step is to specify the name of the file of commands in the `ABORT_FILE` parameter of a `SET_PROGRAM_ATTRIBUTE` or `EXECUTE_TASK` command. Use the former command if you want to specify an `ABORT_FILE` to be used for all subsequent executions in the terminal session. Use the latter command to specify an `ABORT_FILE` to be used only for a particular execution.

Before executing your program, be sure that debug mode is off. (Include the `DEBUG_MODE=OFF` parameter on the `SET_PROGRAM_ATTRIBUTE` or `EXECUTE_TASK` command.) Then begin execution of your program in the usual way. Execution will proceed normally unless an execution error occurs. If an execution error occurs, Debug gets control, executes the commands in the `ABORT_FILE`, and returns control to the system, which issues an error message.

You can include SCL commands in the `ABORT_FILE` as well as Debug commands. This capability makes it possible for you to write sophisticated error handling procedures. SCL procedure writing is covered in the SCL Language Definition manual.

We now consider an example of the use of an ABORT\_FILE. In the example, an ABORT\_FILE is created for a program that contains an error. The ABORT\_FILE contains Debug commands to display the values of all the variables in the program. Before the program is executed, a SET\_PROGRAM\_ATTRIBUTE command is entered to turn off debug mode and to specify the name of the ABORT\_FILE. The PRESET\_VALUE=INFINITY parameter is also specified. Specifying a memory preset other than zero is recommended so that uninitialized variables are readily apparent when displayed by Debug commands. In the terminal dialog, note that the uninitialized variables are displayed as fields of asterisks. (Integer variables are displayed as values so large (19 digits) as to suggest an error.)

The program listing is as follows:

```

1      PROGRAM BUG
2      A = 15.2
3      B = 0.0
4      CALL DIVIDE (A, B, C)
5      PRINT *, A, B, C
6      END

1      SUBROUTINE DIVIDE (R, S, T)
2      T = R/S
3      RETURN
4      END

```

A DIVIDE\_FAULT error occurs when line 2 of subroutine DIVIDE is executed, because the dummy argument S has a value of zero. The ABORT\_FILE for this example, named DBGFILE, is as follows:

```

DISPLAY_PROGRAM_VALUE NAME=A MODULE=BUG
DISPLAY_PROGRAM_VALUE NAME=B MODULE=BUG
DISPLAY_PROGRAM_VALUE NAME=C MODULE=BUG
DISPLAY_PROGRAM_VALUE NAME=R MODULE=DIVIDE
DISPLAY_PROGRAM_VALUE NAME=S MODULE=DIVIDE
DISPLAY_PROGRAM_VALUE NAME=T MODULE=DIVIDE

```

We now execute the program as shown in the following terminal dialog. When the error occurs, the Debug commands are executed and the program terminates.

```
setpa debug_mode=off abort_file=dbgfile preset_value=infinity
lgo
DEBUG
--DEBUG: program terminated by calling abort at M=DIVIDE L=3
a = 15.2
b = 0.
c = *****
r = 15.2
s = 0.
t = *****
--FATAL-- divide fault at P=0B 30 0A4.
```

The SETPA command turns off debug mode, specifies the ABORT\_FILE, and specifies a memory preset value. The LGO command begins execution. When the error occurs, Debug gets control and executes the commands in DBGFILE. Note that because of the memory preset value, the uninitialized variables C and T are displayed as fields of asterisks. Control then passes to the system, which prints the normal error message.

### Example of a Debug Session

To illustrate a typical Debug session, the following program is executed in debug mode.

```
PROGRAM BUG
A = 15.2
B = 0.0
CALL DIVIDE (A, B, C)
PRINT *, A, B, C
END

SUBROUTINE DIVIDE (R, S, T)
T = R/S
RETURN
END
```



The main program, BUG, initializes two variables and passes them to subroutine DIVIDE. Subroutine DIVIDE divides the first value by the second and returns the result to the main program. The program contains the following error: Variable B is set to zero, which results in a divide by zero.

In the following Debug session, the program is allowed to execute until it terminates with an error.

```

exet file=lgo debug_mode=on <----- Begin the Debug session.
DEBUG <----- Execution suspends at
                    beginning of program.

DB/run <----- Begin execution.

-- DEBUG: divide_fault at M=DIVIDE L=3 B0=20 <--Debug gets control
                    when error occurs.

DB/display_program_value name=r <----- Display the value of R.
r = 15.2

DB/display_program_value name=s <----- Display the value of S.
s = 0.

DB/quit <----- End the Debug session.

-- DEBUG:  QUIT terminated task

```

In the following Debug session, a break is set in subroutine DIVIDE. When execution is suspended there, a new value is supplied for the variable S. The program is then allowed to run to completion.

```

exet file=lgo debug_mode=on <----- Begin the Debug session.
DEBUG
DB/set_break break=b1 line=2 module=divide <--Set a break at line 2
                                         of subroutine DIVIDE.

DB/run <----- Begin execution.

-- DEBUG: break B1, execution at M=DIVIDE L=3 B0=20 <--Debug
                                         gets control.

DB/change_program_value name=s value=+2.0 <---Change value of
                                         S to +2.0.

DB/run <----- Resume execution.

    A=15.2 B=2.0 C=7.6 <----- Program prints values and
                                         runs to completion.

-- DEBUG: program terminated ...

DB/quit <----- End the Debug session.
    
```

## Summary of Debugging Aids

FORTRAN and the NOS/VE operating system provide the following debugging aids:

FORTRAN reference map

HELP command

C\$ Directives

Debug utility

## **FORTRAN Reference Map**

The FORTRAN reference map provides debugging information about all the symbolic names used in a program. Generation of the reference map is controlled by the LIST\_OPTIONS parameter on the FORTRAN command.

To generate a complete map, specify

```
LIST_OPTIONS=(S,A,R,M)
```

When running a program interactively, you should also specify the LIST parameter to designate a file to receive the map.

The two most useful sections of the map are the variables section and the statement labels section. The variables map gives the line numbers where each variable is used, and identifies undefined variables and stray names.

The statement labels map gives, for each label, the numbers of the source lines where the label is defined and referenced. You can use this map to identify missing or unneeded labels.

## **HELP Command**

The HELP (or EXPLAIN\_MESSAGE or EXPM) command provides online descriptions of runtime error messages.

If your program terminates with an error message, simply type HELP. You are then taken directly to the description of that message in the online MESSAGES manual. The description in the MESSAGES manual provides three types of information: a brief explanation of the error, some suggestions for correcting the error, and where to go for further information.

If, after reading the description, you would like even more information, you can go directly to the relevant online manual by entering the command described in the Further Information part of the message description.

HELP provides descriptions only of the most recent error in a job. If you want to look up other error messages, such as those that occurred in a previous job, you can use the following EXPLAIN command to enter the MESSAGES manual and go directly to information about the error:

```
EXPLAIN M=MESSAGES S='condition-code'
```

where condition-code is the 6-digit condition code associated with the error.

## C\$ Directives

The compiler control directives can help you debug a program. The most useful directives are the conditional compilation directives and the listing control directives.

The conditional compilation directives enable you to specify blocks of statements within a program that are to be ignored by the compiler. You can use these directives in conjunction with debug PRINT statements that you have inserted in your program. If you surround the PRINT statements by conditional compilation directives, you can direct the compiler either to compile or to disregard the statements by simply changing the value of a single parameter.

The listing control directives enable you to turn off compiler output listing options for portions of a program. These directives are especially useful for modular programs where you are debugging one module at a time. By surrounding each module by listing control directives, you prevent the compiler from producing a source listing and reference map for the modules you are not currently debugging.

## DEBUG Utility

The Debug utility helps you debug a program during execution. It is easy to use and requires no changes to your program. The following list summarizes the steps you can follow to debug a program under control of Debug.

1. When you compile your program, either specify `DEBUG=NONE` and `OPTIMIZATION=DEBUG`, or omit those parameters since `NONE` and `DEBUG` are the default options.
2. Turn on debug mode by specifying `DEBUG_MODE=ON` on either a `SET_PROGRAM_ATTRIBUTE` command or an `EXECUTE_TASK` command.
3. Begin execution by entering a name call or `EXECUTE_TASK` command that specifies the name of your program. This gives control to Debug and begins the Debug session. Debug then prompts you for a command.
4. Enter one or more `SET_BREAK` commands to provide for getting control during execution of the program. The command

**SET\_BREAK NAME=name LINE=line MODULE=module**

sets a break that will suspend execution when the specified line is reached. Alternatively, you can enter the command

**SET\_STEP\_MODE**

to suspend execution at the beginning of each line of the program.

5. Enter a RUN command to begin execution of the program. The program executes until a break is encountered. Then execution suspends and Debug gives control to you.
6. Enter commands to debug the program as desired. Some useful commands are:

**DISPLAY\_PROGRAM\_VALUE VARIABLE=var  
MODULE=module**

Displays the value of a variable or array element.

**CHANGE\_PROGRAM\_VALUE VARIABLE=var  
MODULE=module**

Assigns a new value to a variable or array element.

**DISPLAY\_BREAK ALL**

Lists the breaks currently set in the program.

**DELETE\_BREAK NAME=name1, . . . , namen**

Deletes breaks from a program.

**DISPLAY\_CALL**

Displays a subroutine traceback list.

**DISPLAY\_DEBUGGING\_ENVIRONMENT**

Displays status information about the Debug session.

7. Resume program execution by entering a RUN command. The program will execute until either another break is encountered or the program terminates.
8. Repeat steps 6 and 7 until you are ready to end the Debug session. Then type QUIT to end the session and return to system command mode.



# Introduction to FORTRAN

## Input/Output

---

3

NOS/VE FORTRAN provides a wide variety of input/output methods. Each method is designed for a specific purpose. You can use the information presented in this chapter to gain an understanding of the types of input/output and to help you decide which method is best suited to your needs.

Keep in mind that the purpose of this discussion is to provide an overview of FORTRAN input/output and to help you compare the available methods. The discussion does not provide the detailed specifications you need to effectively use the methods. Before using a particular method, you should read the detailed description of the method in the FORTRAN Language Definition manual.

### Basic Concepts

Input/output is the process of transferring data between the computer's memory and a file. A file is a collection of related data that is identified by a unique name. Every file begins at a boundary known as the beginning-of-information (abbreviated BOI) and ends at a boundary known as the end-of-information (abbreviated EOI).

Data exists on a file in units called records. Generally, a record is the amount of data read or written by a single input or output statement.

Disk files are generally used for data that is to be stored over a period of time. You can also read data from, or write data to, a terminal (interactive jobs), and you can write data to a printer (batch jobs). From the standpoint of a FORTRAN program, terminals and printers are considered to be files, although those devices do not store data as a disk file does.

The type of output discussed in this chapter is called external input/output because data is transferred between memory and an external device (disk, terminal, or printer).

FORTRAN also provides a type of input/output called internal input/output. Internal input/output is not really input/output in the usual sense, because no external devices are used. This type of input/output uses READ and WRITE statements to transfer and reformat data from one area of memory to another. Internal input/output is not discussed in this chapter, but is described in detail in the FORTRAN Language Definition manual.

## Opening Files

Before you reference a file, the file must be opened. The open process prepares the file for input or output and establishes various properties of the file. If you open a file that does not already exist, the file is created.

For some types of input/output, you can either open the file yourself (using an OPEN statement) or you can allow FORTRAN to open the file for you. The OPEN statement enables you to specify various properties of the file. If you don't specify an OPEN statement, the file is automatically opened the first time you reference it in an input/output statement, and FORTRAN provides default values for the file properties.

For other types of input/output, you must explicitly open the file before reading or writing it.

File properties that you can declare on an OPEN statement include the file name, an input/output unit to be associated with the file, and the record length. The OPEN statement is described in detail in the FORTRAN Language Definition manual.

You can also declare file properties on a SET\_FILE\_ATTRIBUTE command. This command allows you to declare many file attributes that cannot be declared in an OPEN statement. You can specify a SET\_FILE\_ATTRIBUTE command before you begin execution of your program, or you can execute the command inside your program through an SCLCMD call. (Both of these commands are described in the FORTRAN Language Definition manual.) However, the OPEN statement is sufficient more most FORTRAN input/output operations.

## Input/Output Units and Unit/File Association

All input/output operations involve the transfer of data between memory and a file. Thus, in every input or output statement, you must indicate the file to be involved in the data transfer. This is done by means of a unit number.

As part of the opening process, all files referenced in a FORTRAN program become associated with a unit number. In an input or output statement, you generally don't specify a file name directly. Instead, you specify a unit number. The operation is then performed on the associated file.



If you open a file yourself (by specifying an OPEN statement) you can establish the association between a file and a unit number. For example, the statement

```
OPEN (UNIT=5, FILE='DOG')
```

associates unit 5 with a file named DOG. Thus, when unit number 5 is referenced in a subsequent input or output statement, file DOG is read or written. For example, the statement

```
READ (UNIT=5, FMT=99) A, B
```

reads two values from file DOG.

If you omit the OPEN statement, thus allowing the open to be done automatically, the unit number is associated with a file having a default name provided by FORTRAN. The default name is formed by prefixing the unit number with the characters TAPE. Thus, in the previous READ statement, assuming no OPEN statement was specified, a file named TAPE5 would be used.

The close process performs certain operations that are required before the file can be referenced in another program. As with the open process, you can either close a file yourself (using a CLOSE statement) or you can allow the file to be closed automatically when the program terminates. Specifying a CLOSE statement provides you with some additional options, although in most cases you can allow the close to be done automatically.

## File Attributes

Every file used under NOS/VE has an associated set of file attributes. These attributes define the file structure and certain processing options associated with the file. A file's attributes are established when the file is created. Some of these attributes are permanent for the life of the file, while others can be changed.

For most types of input/output performed in a FORTRAN program, you don't need to worry about file attributes, because FORTRAN provides an appropriate set of default attributes. The notable exception is the set of file interface subprograms, which require you to know a great deal about file attributes.

## Types of FORTRAN Input/Output

FORTRAN input/output can be classified in the following three ways:

According to whether or not the data is formatted (and if so, according to whether the format is user-specified or provided by FORTRAN).

According to how records on the input or output file are accessed.

According to whether or not the method is a standard ANSI feature.

The formatted methods are used for data that is to be viewed in some way (printed or displayed at a terminal). You can use FORTRAN-supplied default formatting, or you can specify your own detailed format specifications.

The unformatted methods are used for data that will not be viewed. Generally, unformatted input/output is used for files that are to be stored and used as input to other programs.

The two methods of access are sequential access and random access. In sequential access files, records must be read in the order in which they were written. Random access files allow you to read particular records directly, without sequentially searching for the records.

NOS/VE FORTRAN provides all of the input/output methods defined in the 1978 ANSI standard document. Several CDC-unique methods are also provided but should be used only in special circumstances.

A word of caution: Do not try to mix different types of input/output on the same file. Doing so usually results in a runtime error and may destroy the contents of the file.

## Overview of Input/Output Methods

The categories of FORTRAN input/output are as follows:

- **Formatted Input/Output Methods.** These include READ and WRITE statements with format specifications, list directed READ and WRITE statements, and namelist READ and WRITE statements.
- **Unformatted Input/Output Methods.** These include the unformatted READ and WRITE statements and the BUFFER IN and BUFFER OUT statements.
- **Random Access Methods.** These include the direct access READ and WRITE statements, the mass storage input/output subroutines, and the file interface subprograms.

## Formatted Input/Output Methods

Formatted input/output is used for data that is to be printed or viewed in some way. On input, data in the form of a string of characters is read, converted to an internal representation, and stored in memory. On output, values in memory are converted to strings of characters and printed (or written to a file or displayed at the terminal).

NOS/VE FORTRAN provides three ways of doing formatted input/output:

- List Directed Input/Output. This is the easiest method. It uses default formatting.
- Namelist Input/Output. This method also uses default formatting, but you specify a group name instead of an input/output list. This method is non-ANSI.
- Formatted Input/Output with Format Specification. With this method, you specify the formatting to be done. (This method is often referred to simply as formatted input/output.)

### List Directed Input/Output

List directed input/output is the easiest to use of the formatted input/output methods. With list directed input/output, you do not need to specify the formatting to be performed. Instead, a FORTRAN-defined default format is used. List directed input/output is recommended for applications where the actual format of the data is not important.

The list directed output statements are READ, WRITE, and PRINT, with a \* substituted for the format specification. (A PUNCH statement is also provided, but it is seldom used.) The simplest forms of these statements are:

**READ (UNIT=*u*,FMT=\*) list**

**WRITE (UNIT=*u*, FMT=\*) list**

**PRINT \*, list**

where *u* specifies the file to be read or written, \* selects list directed formatting, and list specifies the items to be read or written.

List directed input data has a much freer form than formatted input data. You simply specify a string of values using any of the valid FORTRAN forms for constants, and separate the values by commas or blanks. The following examples show typical list directed input records:

3.5, 753.141, 100

Two real numbers and an integer number, separated by commas.

6.78 (4.1,10.0) 'ABC'

A real number, a complex number, and a character string. The values are separated by blanks.

List directed output values have the same forms as FORTRAN constants. However, a long string of list directed output values can sometimes be difficult to read.

For example, the following READ statement reads character data from unit 3, converts it to internal format, and stores it in memory:

```
CHARACTER*5 STR
READ (UNIT=3,FMT=*) AVAL, BVAL, IFG, STR
```

An example of an input record for this READ statement is:

100.0, 25.0, 50, 'FIRST'

The READ statement reads, converts, and stores the following values:

100.0 is stored in the real variable AVAL

25.0 is stored in the real variable BVAL

50 is stored in the integer variable IFG

FIRST is stored in the character variable STR

The following statements define some variables of varying types and print the values using a list directed PRINT statement. The variable values are converted from internal representation to characters according to a default format, and printed.

```
LOGICAL L
CHARACTER B*3
COMPLEX C
L = .TRUE. <-----a logical value (will be printed as T)
C = (-14,3.55) <-----a complex number
B = 'DOG' <-----a character string
X = 3.14159 <-----a real number
PRINT *, L, C, B, X
```

The output record is:

```
T(-14.,3.55)D0G3.14159
```

Note that the output values are not separated by commas or blanks.

## **Namelist Input/Output**

Namelist input/output is similar to list directed input/output in that you do not supply a format specification. Namelist input/output uses a default format that is consistent with the formats of the FORTRAN constants.

The main difference between namelist and list directed input/output is that with namelist, you do not specify a list of variables to be read or written. Instead, you specify the name of a namelist group that contains all the variables and arrays you want to read or write. The namelist input/output statement then transfers all the items in the specified group.

In order to use namelist input/output, you must first define a namelist group. This group should contain all the variables and arrays that you want to read or write using a single READ or WRITE statement. You define a namelist group by specifying a NAMELIST statement, which has the general form:

```
NAMELIST /name/list
```

where name is a name you assign to the group, and list is a list of variables and arrays to be in the group.

Data to be read by a namelist READ statement must have the form of a namelist input group. The input group specifies the name of each variable (or array) and the value to be assigned to the variable or array. The general form of a namelist input group is:

```
$group-name variable=value, . . . , variable=value $END
```

The dollar sign preceding group-name must appear in character position 2. You can continue a namelist group over successive lines.

The following example shows a simple namelist group:

```
$AGRP VAL(1)=0.456, VAL(2)=7.89, IVAL=100 $END
```

The group is named AGRP, and specifies values for array elements VAL(1) and VAL(2), and the variable IVAL.

To read or write the items in a namelist group, you simply use a READ, WRITE, or PRINT statement that specifies the group name. The general forms of the namelist READ and WRITE statements are:

**READ (UNIT=*u*, FMT=*name*)**

**WRITE (UNIT=*u*, FMT=*name*)**

where *u* is the unit to be read or written and *name* is a namelist group name.

Namelist input/output is especially useful for programs that repeatedly read or write the same items. With namelist, you don't need to list all the items for each input/output operation.

The main disadvantage of namelist input/output is that it is not a standard ANSI feature. Therefore, if you want your program to be portable to other systems, you shouldn't use namelist. Also, the namelist input data format is less convenient than the list directed format, since you must specify each variable and array name and include \$group-name and \$END.

The following program illustrates a namelist READ and WRITE.

```
PROGRAM NMLST

C Define two namelist groups.
  NAMELIST /INGRP/ R,T
  NAMELIST /OUTGRP/ R,T,D

C Read the group named INGRP.
  READ (UNIT=5, FMT=INGRP)
  D = R*T

C Write the group named OUTGRP.
  WRITE (UNIT=6, FMT=OUTGRP)
  END
```

An example of an input group for this program is:

```
$INGRP R=55.0, T=2.5 $END
```

The output written to unit 6 is:

```
$OUTGRP
R = .55E+02
T = .25E+01
D = .1375E+03
$END
```

Output values are stacked vertically and are in a format that can be read by a subsequent namelist READ. Note that E format is used for real values.

## Formatted Input/Output With Format Specification

This is the most powerful and flexible type of formatted input/output, but also the most difficult to use. With this type of input/output, you explicitly specify the formatting to be done during the input or output operation.

The simplest forms of the formatted input/output statements are:

**READ (UNIT=*u*, FMT=*label*) list**

**WRITE (UNIT=*u*, FMT=*label*) list**

where *u* identifies the unit to be read or written, *label* is the label of a format statement, and *list* is a list of variables, arrays, or array elements to be read or written.

A PRINT statement is also provided. The PRINT statement can be used instead of the WRITE statement to print data (batch jobs) or display data at the terminal (interactive jobs).

The FORMAT statement (identified by the FMT= parameter) specifies the formatting to be done.

Some examples of formatted READ and WRITE statements with format specifications are as follows:

```
READ (UNIT=2,FMT=100) A(1), J, ANSWER
```

This statement reads values from unit 2 into the variables A(1), J, and ANSWER. The values are converted from character format to an internal representation. The type of each conversion is specified by the FORMAT statement labeled 100.

```
WRITE (UNIT=99,FMT=1) TIME, RATE, WAGES
```

This statement writes the values in the variables TIME, RATE, and WAGES to unit 99. The values are converted from internal representation to character format. The type of each conversion is specified by the FORMAT statement labeled 1.

The most common way of specifying a format is by using a FORMAT statement, although there are other ways. The FORMAT statement specifies the conversions to be performed on input or output. The FORMAT statement can also specify editing to be done, such as table headings, tabulation, blank control, printer control, and so forth.

The general form of the FORMAT statement is:

**label FORMAT (descriptor-list)**

The **FORMAT** statement contains a list of edit descriptors that correspond to the list items in the **READ**, **WRITE**, or **PRINT** statement. These descriptors specify how each value is to be converted.

Edit descriptors enable you to select the type of formatting to occur during a formatted input/output operation. In general, you specify one descriptor for each item to be read or written. The particular descriptor you should use depends on the type and format of the item.

To see how edit descriptors work, consider the **F** descriptor. The **F** descriptor is used to read or write real (floating-point) numbers. This descriptor specifies the width (number of characters) of the input or output field and the number of characters to the right of the decimal point. For example,

**F8.3**  
 └─ The field has 3 characters to right of decimal point.  
 └─ The field is 8 characters long.  
 └─ Specifies floating-point conversion.

This example specifies a formatted field 8 characters long, with three characters to the right of the decimal point, one character reserved for the decimal point, and four characters (including + or - sign) to the left of the decimal point.

The following example of a formatted **READ** statement reads values into the variables **I** and **X** according to the **FORMAT** statement labeled 50.

```
READ (UNIT=2, FMT=50) I, X
50 FORMAT (I3, F10.4)
```

The **FORMAT** statement specifies the following two input fields:

**I3** specifies an integer field three characters long.

**F10.4** specifies a floating-point field 10 characters long, with 4 characters to the right of the decimal point and 5 characters to the left. (The decimal point counts as a character.)

A typical input record for this **READ** statement is as follows:

bb8bb456.1760 (b indicates a blank character.)

The number 8 is read into **I**, and the number 456.1760 is read into **X**.



The following sequence writes two numbers to unit 2. The numbers are converted from internal format to characters as specified in the FORMAT statement.

```

      J = 24
      VALUE = 3.14159
      .
      .
      .          WRITE (UNIT=2,FMT=99) J, VALUE
99  FORMAT (1X,'INDEX IS ', I1, ' RESULT IS ', F8.3)

```

These statements produce the following output:

```

      INDEX IS 24  RESULT IS   3.14159

```

The variable J is converted and written according to the descriptor I1, and the variable VALUE is converted and written according to the descriptor F8.3.

### Examples Comparing the Formatted Input/Output Methods

Following are three versions of a short program that reads several input values and writes those same values. The first version uses list directed input/output, the second uses namelist input/output, and the third uses formatted input/output with format specifications. The purpose of these examples is to help you compare the three methods of formatted input/output.

The examples are written to perform interactive input and output (that is, data is read and written at the terminal). Each example is accompanied by a sample terminal dialog, showing input and output.

**List Directed Input/Output Example**

Source Listing:

```

PROGRAM LSTDIR
CHARACTER NAME*10

C List directed PRINT statement.
  PRINT *, 'ENTER NAME, ACCOUNT, AND AMOUNT'

C List directed READ statement.
  READ (UNIT=*, FMT=*) NAME, IACCNT, AMT

C List directed PRINT statement.
  PRINT *, NAME, IACCNT, AMT
END

```

Terminal Dialog:

```

lgo <----- Begin execution.

ENTER NAME, ACCOUNT, AND AMOUNT
? 'smith' 00324 1050.00 <----- User input. Spaces are used as
                                separators. Note that character
                                strings are enclosed in
                                apostrophes.

SMITH    324 1050. <----- Program output, showing list
                                directed output format. Note that
                                leading zeros were dropped from
                                account number, and zeros to
                                right of decimal point were
                                dropped from amount.

```

## Namelist Input/Output Example

Source Listing:

```

PROGRAM NMLST
CHARACTER NAME *10

C Define namelist group.
  NAMELIST /GRP/ NAME, IACCNT, AMT

C Use list directed PRINT statement to print prompt.
  PRINT *, 'ENTER NAME, ACCOUNT, AND AMOUNT'

C Namelist READ and WRITE statements.
  READ (UNIT= *, FMT=GRP)
  WRITE (UNIT= *, FMT=GRP)
  END
    
```

Terminal Dialog:

```

lgo <----- Begin execution.

      ENTER NAME, ACCOUNT, AND AMOUNT <---- Program prints
                                           prompt.

? $grp name='smith', iaccnt=00325, amt=1050.00 $end <-- Enter
                                           input.

$GRP
NAME   = 'SMITH   ',
IACCNT = 325,
AMT    = .1050E+04,
$END
    
```

<----- Namelist output.

In namelist input, data is free form, but you must specify the name of each variable or array. The input record must begin with \$group-name and end with \$END. Output values are stacked vertically, and include the variable or array name. Note that exponential (E) format is used for real values. Note also that the output is in a format that can be read by a namelist READ.

### Example of Formatted Input/Output With Format Specification

Source Listing:

```

PROGRAM FMTSPEC
CHARACTER NAME*10

C Print prompt for user input.
PRINT 100
100 FORMAT (' ENTER NAME, ACCOUNT, AND AMOUNT')

C Read input values according to FORMAT statement 101.
READ (UNIT=*, FMT=101) NAME, IACCNT, AMT
101 FORMAT (A10, I5, F8.2)

C Print table headings according to FORMAT statement 102.
PRINT 102
102 FORMAT (' NAME', 6X, 'ACCOUNT', 3X, 'AMOUNT', /)

C Print output values according to FORMAT statement 103.
PRINT 103, NAME, IACCNT, AMT
103 FORMAT (1X, A10, 1X, I5.5, 3X, '$', F8.2)
END
    
```

Terminal Dialog:

```

lgo <----- Begin execution.
ENTER NAME, ACCOUNT, AND AMOUNT <-- Prompt for user input.
? smith      00324 1050.00 <----- User input.
NAME        ACCOUNT   AMOUNT   }
SMITH       00324    $ 1050.00 } <-- Program output.
    
```

Note that the user input must be within the fields specified by the FORMAT statement. Note also that by using appropriate edit descriptors, you can "dress up" the output so that it is more meaningful. In this example, edit descriptors were used to print table headings, place leading zeros in the account number, print a dollar sign, and print trailing zeros in the amount.

## Unformatted Input/Output Methods

In unformatted input/output, data is transferred directly between memory and a file without any conversions or editing. The data is in its internal representation, and generally cannot be printed or viewed at a terminal.

Unformatted input/output is faster than formatted input/output because the time required for formatting and editing is eliminated. Unformatted input/output is generally used for data that will not be printed or displayed at a terminal. For example, if you want to write data to a file, and subsequently read that file in another program, you should consider using unformatted input/output.

The methods of unformatted input/output are:

- Unformatted READ and WRITE Statements. This is the preferred method.
- BUFFER IN and BUFFER OUT Statements. This method is not recommended and is discussed only briefly.

## Unformatted READ and WRITE Statements

Unformatted READ and WRITE statements provide a way of transferring data between memory and a file without converting or editing the data.

The unformatted READ and WRITE statements are similar to formatted READ and WRITE statements, except that they do not specify a FORMAT label. The simplest forms of the unformatted READ and WRITE statements are:

**READ (UNIT=*u*) list**

**WRITE (UNIT=*u*) list**

where *u* identifies the input/output unit, and list is a list of variables or arrays to be read or written.

For example, the following unformatted READ statement reads data from unit 9 into arrays ARR and BRR. The data is stored exactly as it appears on the file.

```
DIMENSION ARR(100), BRR(250)
.
.
.
READ (UNIT=9) ARR, BRR
```

The following unformatted WRITE statement writes data from array XDAT and variables T and V to the file associated with unit 9. The data is written in exactly the same form as it appears in memory.

```
DIMENSION XDAT(125)
.
.
.
WRITE (UNIT=9) XDAT, T, V
```

### **BUFFER IN and BUFFER OUT Statements**

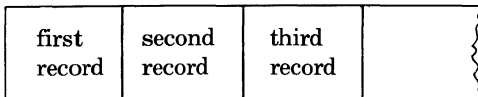
The BUFFER IN and BUFFER OUT statements provide an alternate method of unformatted input/output. BUFFER IN and BUFFER OUT exist mainly for compatibility with other versions FORTRAN, and their use is not recommended. For unformatted input/output, you should use unformatted READ and WRITE wherever possible.

There are three major differences between BUFFER statements and unformatted READ and WRITE:

- Unformatted READ and WRITE statements specify a list of items to be read or written. BUFFER statements specify the first and last words of a block of memory.
- Unformatted READ and WRITE statements can transmit character data, whereas BUFFER statements cannot.
- Unformatted READ and WRITE statements are standard ANSI statements, whereas BUFFER statements are not. Therefore, any program that contains BUFFER statements is not portable to other systems.

### **Random Access Methods**

Up to now, we have assumed that the files you are reading and writing are sequential access files. On a sequential access file, all records are physically stored in the same order in which they were written. The records can be read only in that order. A sequential file has the following general form:



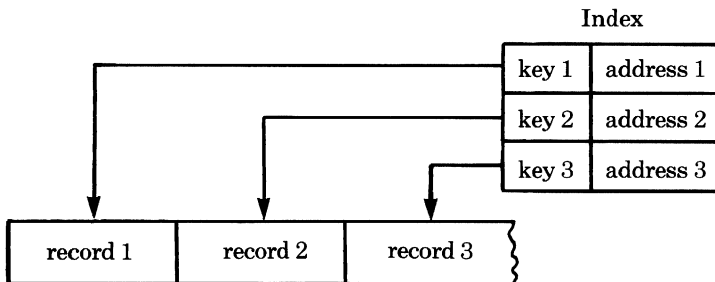
On a sequential file, you can access a particular record only by reading all physically preceding records, until you reach the desired record. For instance, in order to read the third record, you must read the first and second records (or bypass them using the SKIP statement). Of course, the additional READ or SKIP operations require additional time.

Sequential files are easy to use, and are fine for most purposes. But in programs where you want to read or write specific records without reading or writing all the preceding records, sequential files can be slow and inconvenient because they require you to sequentially search for the desired record.

A much faster method of retrieving specific records on a file is provided by random access files. On a random access file, each record has a record key associated with it. To read or write a particular record, you simply specify the key for that record. You are then taken directly to the desired record. Thus, the need for sequentially searching for a particular record is eliminated.

A random access file has an index associated with it. The index contains the key and disk address of each record on the file. To access a particular record, you specify the record key in a random access input/output statement. The software looks up the address of the record in the index, and then goes directly to that record.

The following diagram illustrates a random access file:



You should consider a random access method whenever you want to create a file in which records will be accessed in an order different from the order they were created in.

NOS/VE FORTRAN provides the following ways of doing random access input/output:

- FORTRAN Direct Access Files. This is the recommended method. It is the easiest to use and it is a standard ANSI method.
- File Interface Subprograms. This method provides the most options and flexibility, but it is the most difficult to use. It is a CDC extension.
- Mass Storage Input/Output Subroutines. The difficulty and flexibility of this method is somewhere between the preceding methods. This method is a CDC extension.

## **FORTRAN Direct Access Files**

FORTRAN direct access files provide an easy way of doing random access input/output. These files are read and written using standard FORTRAN READ and WRITE statements.

To create a direct access file, you must specify an OPEN statement that contains the ACCESS='DIRECT' specifier. You must also specify the record length of the file using the RECL specifier in the OPEN statement. (The record length is the number of words (unformatted data) or characters (formatted data) to be read or written. (Refer to the FORTRAN Language Definition manual for an explanation of how to calculate the record length of a direct access file.)

For example, the statement

```
OPEN (UNIT=2, FILE='DFILE', ACCESS='DIRECT', RECL=120)
```

creates an unformatted direct access file named DFILE with a record length of 120 words.

If you omit the ACCESS='DIRECT' specifier from the OPEN statement, the file is automatically created as a formatted sequential access file.

Each record in a direct access file is identified by a record number. You assign the record number the first time you write the record. The record number you assign is permanently associated with the record. Then, when you want to read or rewrite the record, you simply specify the record number in the input/output statement. In general, you assign the number 1 to the first record written, 2 to the second record, and so forth.

You read and write direct access files using standard FORTRAN READ and WRITE statements. But you must include the REC parameter in these statements. REC specifies the record number of the record you want to read or write.



You can use direct access files for either formatted or unformatted input/output. If you want to use a direct access file for formatted input/output, you must specify `FORM='FORMATTED'` in the `OPEN` statement that creates the file. Otherwise, the file is automatically created for unformatted input/output.

The following sequence opens an (unformatted) direct access file and writes a single record:

```
OPEN (UNIT=2, FILE='NEWFL', ACCESS='DIRECT', RECL=120)
.
.
.
WRITE (UNIT=2,REC=8) A, B
```

The file is named `NEWFL` and has a record length of 120 words. The `WRITE` statement writes the variables `A` and `B` to record number 8 of the file.

The following sequence opens and reads a formatted direct access file:

```
OPEN (UNIT=6, FILE='AAA', ACCESS='DIRECT', FORM='FORMATTED',
+ RECL=100)
.
.
.
DO 5 I=2,10,2
    READ (UNIT=6, FMT=50, REC=I) XARR
50  FORMAT (10F12.4)
5  CONTINUE
```

The `DO` loop reads records 2, 4, 6, 8, and 10.

The following example illustrates a simple, but typical, direct access application.

Program `UPDATE` is an application that might be found at a bank. The purpose of this program is to process customer withdrawals and deposits. The program assumes the existence of an unformatted direct access master file named `ACCNPTS`, containing each customer's name, account number, and account balance. The contents of this file are as follows:

<u>word 1</u> (Name)	<u>word 2</u> (Account No.)	<u>word 3</u> (Balance)
SMITH	00324	1000.00
JONES	70516	2500.00
CLARK	15922	10000.00
STEVENS	03304	650.00
WILSON	21090	125.00

The program interactively updates the master file ACCNTS for each withdrawal or deposit.

The program first requests a customer's account number, the type of transaction (W for withdrawal, D for deposit), and the amount of the transaction. This is done using a list directed PRINT statement.

The program then waits for the customer to type in the requested values. The values are read using a formatted read statement. Note that the customer's input must conform to the FORMAT statement labeled 11.

After the customer enters the requested values, the program searches array NARRAY to locate the customer's account number. When the account number is located, the value of the DO index I is the value of the customer's record number on the master file. (Remember that each record on a direct access file is identified by a record number.) If the account number cannot be found, an informative message is displayed and the program terminates.

The program then uses the record number in a direct access READ statement to read the customer's name, account number, and balance from the master file. If the customer requested a withdrawal, the transaction amount is subtracted from the balance. If the customer requested a deposit, the amount is added to the balance. In either case, the new balance is displayed and the updated record, containing the new balance, is written to the master file.

The source for program UPDATE is as follows:

```

PROGRAM UPDATE
CHARACTER NAME*10, NARRAY(5)*5, NACCT*5, T*1
DATA NARRAY /'00324', '70516', '15922', '03304', '21090'/

OPEN (UNIT=2, FILE='ACCNTS', ACCESS='DIRECT', RECL=3)

C Prompt the customer for account number, transaction type,
C and transaction amount. Then read the input values.
PRINT *, 'ENTER ACCOUNT NUMBER, W OR D, AND AMOUNT.'
READ (UNIT=*, FMT=11) NACCT, T, AMOUNT
11  FORMAT (A5, 1X, A1, F10.2)

C Find customer's account in master file. Determine whether
C transaction withdrawal or deposit, then update account.
DO 15 I=1,5
  IF (NACCT .EQ. NARRAY(I)) THEN
    READ (UNIT=2, REC=I) NAME, N, BAL
    IF (T .EQ. 'W') THEN
      BAL = BAL - AMOUNT
    ELSE
      BAL = BAL + AMOUNT
    ENDIF
  ENDIF

C Print customer's name, account number, and new balance.
PRINT 22, N, BAL
22  FORMAT ('ACCOUNT NO. IS ',A5,'. NEW BALANCE IS $',F10.2)
WRITE (UNIT=2, REC=I) NAME, N, BAL
STOP
ENDIF
15  CONTINUE

C
PRINT *, 'INVALID ACCOUNT NUMBER.'
END

```

Following is an example of terminal dialog for program UPDATE:

```
lgo <-----Begin first
                                execution.

ENTER ACCOUNT NUMBER, W OR D, AND AMOUNT. <-----Prompt for
                                user input.

? 00324 w 100.00 <-----User input.

ACCOUNT NO. IS 00324. NEW BALANCE IS $ 900.00 <----Program
                                output.

lgo <-----Begin second
                                execution.

ENTER ACCOUNT NUMBER, W OR D, AND AMOUNT.
? 15922 d 550.

ACCOUNT NO. IS 15922. NEW BALANCE IS $ 10550.00
```

## File Interface Subprograms

The file interface subprograms provide a powerful and flexible method of random access input/output. Some of the advantages of file interface input/output are:

- You can define, access, and change the attributes of a file.
- You can process files having a variety of different attributes.
- You can process all three of the NOS/VE record types (variable, fixed, and undefined). The FORTRAN direct access method allows only fixed-length records.
- You can read or write sequentially as well as randomly.
- You can write routines that are executed when certain errors occur during an input or output operation.

The file interface subprograms are somewhat more difficult to use than the other FORTRAN input/output statements, mainly because you must have a basic understanding of the concepts and structure of the files processed by the subprograms. (These files are known as indexed sequential files.)

A second disadvantage of the file interface subprograms is that they are non-ANSI. Thus, any program that uses them is not portable to other systems.

Because using the file interface subprograms requires extensive background information, they are not explained here. Instead, brief summaries of the subprogram calls and of the file attributes of indexed sequential files are presented in this chapter. Refer to the FORTRAN Language Definition manual for a detailed description of file interface input/output.

<u>Operation</u>	<u>CALL Statement</u>
Define file attributes	CALL FILEIS, CALL STOREF
Retrieve file attributes	IFETCH function reference
Open a file	CALL OPENM
Position a file to a specific record	CALL SKIP, CALL STARTM
Rewind a file	CALL REWND
Read a record	CALL GET for random read, CALL GETN for sequential read
Write a record	CALL PUT, CALL PUTREP
Replace a record	CALL REPLC, CALL PUTREP
Delete a record	CALL DLTE
Close a file	CALL CLOSEM

The files processed by the file interface routines are known as indexed sequential files. Like all other files under NOS/VE, each indexed sequential file is described by a set of file attributes. A major advantage of the file interface calls is that you have complete control over the file attributes. (The obvious disadvantage is that you must understand what all these attributes mean, and when and how they must be defined.) The other FORTRAN input/output methods provide much less control over these attributes.

Indexed sequential files have about 40 different attributes. The following table describes a few of these attributes, in order to give you an idea of the file characteristics over which you have control.

<u>Attribute</u>	<u>Description</u>
COLLATE_TABLE_NAME	Specifies the name of a collation table to be used for comparing character record keys.
ERROR_EXIT_NAME	Specifies the name of a routine to be executed if an input/output error occurs.
FILE_POSITION	Returns a value indicating the position of the file after the last file operation.
KEY_ADDRESS	Defines the record key.
KEY_TYPE	Specifies the type of the record key.
OPEN_POSITION	Specifies where the file is to be positioned when it is opened.
RECORD_LENGTH	Specifies the length of the record being read or written.
RECORD_TYPE	Specifies the type of records in the file (types are variable length, fixed length, and undefined length).

### **Mass Storage Input/Output Subroutines**

The mass storage input/output (MSIO) subroutines provide a third alternative for performing random access input/output. The MSIO subroutines were widely used before the advent of the standard ANSI FORTRAN direct access files. Now, however, the latter method is preferred for most applications. MSIO exists mainly for compatibility with earlier versions of FORTRAN, and in most cases, you should first consider either direct access files or the file interface subprograms.

The MSIO subroutines do provide some advantages over FORTRAN direct access files:

- You can use both fixed-length and variable-length records (direct access allows only fixed-length records).
- You can define a record key, which in many cases is more meaningful than the record number used by the FORTRAN direct access method (more on this later).
- You can define more than one key for a given record.

The file interface routines also offer these advantages, but they are more difficult to use than the MSIO routines.

The following table summarizes the mass storage subroutines:

<u>Operation</u>	<u>Subroutine Call</u>
Open a file.	CALL OPENMS
Write or replace a record.	CALL WRITMS
Read a record.	CALL READMS
Close a file.	CALL CLOSMS
Define an alternate index array. (Enables you to identify a record by more than one key).	CALL STINDEX

A major advantage of the MSIO routines over FORTRAN direct access files is that the MSIO routines provide more flexibility in defining a record key.

Remember that a record key is a value, which you select, that is used to identify the records in a file. To reference a particular record, you simply specify the record's key in the appropriate input or output statement, and the associated record is read or written.

Because of the way the MSIO routines associate record keys with the actual locations of the records on disk, you can assign record keys that are more meaningful than the record numbers used with FORTRAN direct access files. For example, you can use a value such as a person's name or a social security number to identify a record. You can also define alternate record keys, which allow a record to be identified by more than one key. With FORTRAN direct access files, you are limited to using the record number as the record key (the first record written is numbered 1, the second record is numbered 2, and so forth).

An example of mass storage input/output is as follows. Program MSIO reads a record from an existing mass storage file, modifies the record, and rewrites the record.

```

PROGRAM MSIO
  DIMENSION INDEX(10), X(100)

C  Open a mass storage file named TAPE3.  INDEX is the index
C  array, and the record length is 10 words.
  CALL OPENMS (3, INDEX, 10, 1)

C  Read the record having the key APPLE into array X.
  CALL READMS (3, X, 100, 'APPLE')

C  Modify the record.
  X(10) = X(1) + 1.0

C  Rewrite the record.
  CALL WRITMS (3, X, 100, 'APPLE')

C  Close file TAPE3.
  CALL CLOSMS (3)
  END
  
```

## Selecting an Input/Output Method

By now you should have a basic understanding of the available input/output methods and what their differences and similarities are. We now compare the methods and discuss their advantages and limitations in order to help you decide which method to use.

Before you select an input/output method, you should have some idea of what your needs are. As a starting point, you might consider the following questions:

Does the output need to be formatted? (If so, what kind of formatting?)

Which access method should I use? (You have two choices: sequential access and random access)

Should I use a standard ANSI method? (In most cases, you can and should use a standard ANSI method)

These questions are addressed below.



## Does the Output Need to be Formatted?

FORTRAN input/output can be divided into two basic types: that which formats data on input or output, and that which does no formatting of data. The formatted methods are used mainly for data that is going to be viewed in some way. If you intend to print the data or read it at a terminal, you must use one of the formatted methods.

If, however, you do not intend to view the data (for example, if you just want to store the data or possibly use it for input to another program), then you can use unformatted input/output. The unformatted methods are a little faster than the formatted methods because no converting of data is done during the transfer. But remember that unformatted data can't be printed or displayed at a terminal.

The formatted operations perform conversions and editing during the data transfer. On a formatted output operation, data is converted from its internal form (the way it is represented in memory) to a string of characters. On a formatted input operation, data in the form of a string of characters is read and converted to an internal representation and stored in memory.

The formatted methods are:

- Formatted with format specification

- List directed

- Namelist

Format specifications are the most flexible, but are the most difficult to use. This type of input/output requires you to write detailed specifications that define the format of the input data and of the output data. Format specifications are ANSI-defined, so your programs will be portable to other systems. If you want complete control over the appearance of the input and output data, use this method.

List directed input/output is the easiest to use of the formatted methods. It is recommended for novice programmers, and for programs where the format of data is not important. List directed input/output uses a default format that is provided by FORTRAN. This default format is somewhat limited, especially on output, where the format may be difficult to read. List directed input/output is a standard ANSI method, and is therefore portable to other systems.

Namelist input/output, like list directed input/output, uses a default format. The format is somewhat more restricted than that used by list directed input/output, but is more legible. A second advantage of namelist is that you do not specify lists of variables and arrays on the READ and WRITE statements. Instead, you specify the name of a namelist group which you have defined earlier in the program. The READ or WRITE statement then transfers all the items that you have assigned to the group.

The main disadvantage of namelist input/output is that it is not a standard ANSI method. Thus, programs using namelist must be modified before being moved to other systems.

If you decide to use an unformatted method, you can choose either unformatted READ and WRITE or BUFFER IN and BUFFER OUT. Both of these methods transfer data between memory and a file without any conversion whatsoever.

There is little no difference in the efficiency and usability of the two methods. A minor difference is that with unformatted READ and WRITE statements, as with other READ and WRITE statements, you specify a list of items to be read or written. With BUFFER statements, you specify the starting location and ending location of a block of memory to or from which data is to be transferred.

Usually, unformatted READ and WRITE are preferred over BUFFER statements. The unformatted READ and WRITE statements are syntactically similar to other READ and WRITE statements, and they are ANSI-defined. The BUFFER statements are nonstandard, and are provided mainly for compatibility with previous versions of FORTRAN. In general, you are discouraged from using them.

## **Which Access Method Should I Use?**

FORTRAN input/output can be classified according to the way data on a file is accessed. The two ways of accessing data on a file are sequential access and random access.

In a sequential access file, records are stored in the order in which they were written and can be retrieved only in the same order.

In a random access file, you can retrieve records in any order, without the need for a sequential search.

Sequential access input/output is the most common method of access and should always be used in programs where you want to read and write records sequentially. (Note that all files created in a FORTRAN program are created as sequential access files, unless you explicitly request otherwise.)

If you want to create a file where records will be accessed in an order different from the order in which they were created, you should use one of the random access methods. The random access methods provide much quicker access to individual records than the sequential access methods. The sequential methods require you to search sequentially for a desired record. The random access methods enable you to go directly to a desired record by specifying a key (or number) associated with the record.

The methods of random access are:

- FORTRAN direct access files

- File interface subprograms

- Mass storage input/output subroutines

FORTRAN direct access is the easiest method to use. It requires no knowledge of internal file structure or file attributes (FORTRAN-supplied defaults are used). It uses standard FORTRAN READ and WRITE statements, and you can perform either formatted or unformatted input/output on direct access files. In addition, it is an ANSI-defined method and can therefore be transported to other systems.

The major disadvantage of FORTRAN direct access is in the use of record keys. Remember that a record key is a value that you assign to a record, and which is used to identify that record. With FORTRAN direct access files, you are limited to using an integer value for the record key. And because of the way the direct access method computes the disk addresses of records, the first record should be numbered 1, the second record numbered 2, and so forth.

If you require a more flexible method, the file interface subprograms provide complete control over the attributes of files. You can define, retrieve, and modify the attributes of a file, and you can process files having many different kinds of attributes.

The file interface routines are much more difficult to use than the FORTRAN direct access method, because you must have a complete understanding of the attributes of your files and of the many processing restrictions and requirements involved with file interface input/output.

The file interface calls should be used only if your application requires you to have nearly complete control over file attributes and processing options. In addition, the file interface subprograms are non-ANSI, thus requiring program modification before moving to another system.

The mass storage subroutines provide a third method of random access. This method ranges in difficulty and flexibility somewhere between FORTRAN direct access and the file interface subprograms.

The mass storage subroutines provide more flexibility and more options than direct access in the selection and use of a record key. In particular, you can define more meaningful record keys, and you can also define alternate record keys. But the mass storage subroutines don't provide as much control as the file interface subprograms.

In general, FORTRAN direct access is preferred over the mass storage subroutines, despite its limitations. The mass storage subroutines exist mainly for compatibility with earlier versions of FORTRAN. In addition, they are non-ANSI and thus reduce a program's portability.

## Should I Use a Standard ANSI Method?

If it is important that your program be portable to other systems, you should use a standard ANSI method. You can select an ANSI method for almost any input/output situation. The standard methods all use variations of the FORTRAN READ and WRITE statements and thus are easy to learn because of their similarity to one another.

The standard ANSI methods are as follows:

- Formatted with format specification (direct or sequential access files)
- List directed (direct or sequential access files)
- Unformatted (direct or sequential access files)

The following methods are nonstandard and should not be used if portability is a concern:

- Namelist
- Buffer
- Mass Storage Subroutines
- File Interface Subprograms

# Summary of FORTRAN Input/Output Statements

The following table summarizes the FORTRAN input/output methods:

<u>Statements</u>	<u>Formatting</u>	<u>Access Method</u>	<u>ANSI?</u>
READ, WRITE with FORMAT specification	Programmer specifies	Sequential or random	Yes
List directed READ, WRITE	Default	Sequential or random	Yes
Namelist READ, WRITE	Default	Sequential	No
Unformatted READ, WRITE	None	Sequential or random	Yes
BUFFER IN, BUFFER OUT	None	Sequential	No
Mass Storage Subroutines	None	Random	No
File Interface Subprograms	None	Random	No



Optimization is the process of altering a source program to make it execute faster. The purpose of optimizing a program is to reduce the cost of executing that program. Always keep in mind that other factors influence the cost of a program besides the amount of computer time required to execute the program. In particular, you should consider the time required to write, debug, and maintain the program. If an optimization technique requires an extensive amount of your time to implement, or if it would make the program more difficult to maintain, the technique may not be worthwhile even though it increases the speed of your program.

## Basic Concepts

You can direct the FORTRAN compiler to produce an optimized object program by specifying (or in some cases, not specifying) certain parameters on the FORTRAN command. (These parameters are discussed later.) In most cases, that's all you need to do. Sometimes, however, you may find it necessary to make your program execute even faster. In those cases, there are usually some simple changes you can make to your source code to make it more efficient.

The FORTRAN compiler is capable of performing a number of useful optimizations. This capability is controlled by the `OPTIMIZATION` parameter. By simply specifying `OPTIMIZATION=HIGH` on the FORTRAN command, you can significantly increase the speed of a program. Thus, in most cases, you don't need to alter your program in order to increase its execution speed. But there are some simple guidelines you can follow that will actually make it easier for the compiler to optimize a program:

- Keep program units small (less than about 100 lines).

- Use a structured style of programming that avoids branching and other complicated logic.

- Check the program for careless errors such as variables that are defined but never used. (The compiler will detect most of these errors.)

Thus, the optimization process actually begins during the writing of a program. By following these guidelines, you can help the compiler produce more efficient object code, and avoid having to rewrite your completed program later.

A note about the `OPTIMIZATION` parameter: Always do your debugging runs with `OPTIMIZATION=DEBUG` (or `LOW`) specified on the `FORTRAN` command (or omit the `OPTIMIZATION` parameter, since `DEBUG` is the default option). This results in faster compilation, which is more important than fast execution during the debugging process.

Once you have debugged your program, you are ready to consider ways to make it run faster. At this point, you should stop and consider whether it's really important to speed up the program. Optimizing a program can take a significant amount of your time. A slight increase in execution speed may not be worth the time required to produce that increase.

Before you begin, a few words of caution:

- Avoid optimizations that make your program more obscure and harder to understand. Such optimizations ultimately make the program more difficult (and thus, more expensive) to maintain or change. (Remember that the purpose of optimizing a program is to reduce the total cost of that program.)
- If you must do optimizations that make a program less clear, be sure you document those optimizations by inserting appropriate comments in the source program. Please be considerate of the programmer who, at some future time, may have to modify the program.
- Avoid spending too much time optimizing a program. Writing, debugging, and maintaining an optimized program often contribute more to the overall cost of that program than the time required to execute it.

Finally, you should be aware that some optimizations may slightly change the results of the program, although the change is usually not significant. You can control these optimizations by specifying certain options on the `FORTRAN` command, as described later in this chapter under `FORTRAN Command Parameters`. The optimizations described in this chapter involve very simple manipulations of the source code. They are not fancy "tricks", and they do not require a knowledge of machine instructions.



## Optimization Techniques

The FORTRAN compiler performs the following general types of optimizations:

- Optimizations that eliminate operations from the code.
- Optimizations that move operations out of loops so that the operations are executed less frequently.
- Optimizations that replace slower operations with faster operations.

The most beneficial changes you can make to your program are those that help the compiler perform these optimizations. The rest of this discussion describes how the compiler performs the above optimizations and explains how you can help the compiler perform them.

## Eliminating Unnecessary Operations

An operation can be eliminated from a program:

- If the result of the operation is not used anywhere in the program.
- or
- If the result of the operation is available even if that operation is not executed.

The following paragraphs discuss the types of unnecessary operations that the compiler will eliminate.

### Eliminating Identity Instructions

An identity instruction is one whose result is the same as one of its operands. The compiler can recognize certain identity instructions and remove the unnecessary operations from them.

For example, the compiler will change the expression

$$Z + (X - Y) + Y$$

to the equivalent form

$$Z + X$$

The change eliminates the unnecessary addition and subtraction of Y.

## Eliminating Dead Instructions

An instruction is dead if it produces a result that is never used. Dead instructions can be safely removed from the program.

An example of a dead instruction is a store into a local variable immediately before a RETURN or END statement. Since the value of the variable can never be used, the compiler can eliminate the instructions that calculate and store the value.

For example, the following subroutine contains dead instructions:

```
SUBROUTINE SUB (X,Y,Z)
  .
  .
  .
  V = X - Y
  RETURN
  END
```

The instructions generated by the statement  $V = X + Y$  are never used and are therefore dead instructions. The compiler will eliminate the entire statement  $V = X + Y$ .

Note that the variable being stored into must be a local variable. That is, it must not appear in a COMMON statement or an argument list.

## Evaluating Constant Subexpressions

A constant subexpression is a subexpression that consists only of constants. At both HIGH and LOW optimization levels, the compiler attempts to evaluate as many constant subexpressions as possible, so that these subexpressions need not be evaluated during execution.

(Remember that because programs are usually executed many more times than they are compiled, it is beneficial to decrease execution time at the expense of increased compilation time.)

For example, when the compiler processes the statement

```
X = 3.5 + 4.0*2
```

the compiler evaluates the expression. The optimized statement is equivalent to

```
X = 11.5
```

You can help the compiler evaluate constant subexpressions by arranging expressions so that constant subexpressions are recognizable. For example, the compiler will not recognize the constant subexpression  $3.14159/2.0$  in the statement

$$W = X * 3.14159 * Y * Z / 2.0$$

But the compiler will recognize the constant subexpression in the statement

$$W = X * Y * Z * 3.14159 / 2.0$$

## Eliminating Unnecessary Stores and Fetches

If a variable is used several times within an instruction sequence, the compiler can sometimes eliminate repeated stores and fetches of the variable by placing the variable value in a register. For example, in the statements

$X = Y + Z$	X is defined
$A = X + B$	X is referenced
$X = X/R$	X is redefined

the compiler does not need to store X after the first assignment statement because X is immediately redefined. The compiler eliminates this unnecessary operation by placing X in a temporary register after the first assignment.

## Eliminating Common Subexpressions

A common subexpression is an expression that occurs more than once in a program unit. In unoptimized code, the expression is evaluated each time it occurs. When optimizing a program, however, the compiler tries to save the result of the expression in a register and to use that result instead of reevaluating the expression.

For example, in the sequence

$$X = A * B * C$$

$$S(A * B) = (A * B) / C$$

the compiler recognizes  $A * B$  as a common subexpression occurring three times. In the optimized code,  $A * B$  is evaluated once and the result is used three times.

When a subexpression contains more than one operator of equal precedence, the compiler may reorder the operations to form common subexpressions. For example, the expression

$$A + B + C$$

would normally be evaluated from left to right (add C to the result of A + B). However, the compiler may evaluate B + C first in order to form a common subexpression.

The reordered form is guaranteed to be mathematically equivalent to the original form, but not necessarily computationally equivalent.

In order to eliminate common subexpressions, the compiler must be able to recognize the subexpressions as the same expression.

For example, in the expression

$$A + B + C + D$$

the following are recognized as subexpressions:

$$A + B \quad A + B + C \quad C + D$$

and so forth. But A + D is not recognized as a subexpression.

Code that might change the value of a subexpression must not appear between occurrences of that subexpression.

For example, in the sequence

```
X = A(2)/B(2) + Q
A(I) = 4.5
Z = A(2)/B(2) + 13.4
```

A(2)/B(2) cannot be treated as a common subexpression because of the possibility that I might be equal to 2, which would change the value of A(2)/B(2).

You can help the compiler recognize and eliminate unnecessary instructions in several ways:

- The most important way is to keep program units short (less than 100 lines) and simple. Avoid complicated logic, and avoid branching.
- Program carefully. Unnecessary instructions are often the result of programming errors.

- Rewrite expressions so that the compiler is more likely to recognize certain subexpressions. For example,

$$A + D + B + C$$

ensures that  $A + D$  will be recognized as a subexpression.

- Use parentheses to ensure a desired grouping of subexpressions. For example,

$$X = C*(A*B)/D$$

$$Y = (A*B)/C$$

ensures that  $A*B$  will be recognized as a common subexpression.

- Arrange your program so that expressions containing subexpressions are as close together as possible. In particular, be sure that code that might change the value of a subexpression does not appear between occurrences of that subexpression.

## Removing Operations From Loops

The techniques of eliminating unnecessary instructions yield the greatest benefit when applied to loops. For example, removing a common subexpression within a loop that is repeated 10,000 times results in a greater time savings than removing the subexpression from straight-line code.

Another optimization technique that the compiler applies to loops is the removing of invariant code.

Invariant code is a sequence of instructions, within a loop, whose operands and result do not change throughout execution of the loop. The compiler removes invariant code from loops whenever possible. For example, in the statements

```
DO 100, I=1,10
  K(I) = J/L + I**2
100 CONTINUE
```

neither  $J$  nor  $L$  changes during execution of the loop. Therefore,  $J/L$  is invariant. During compilation, the compiler removes the invariant instructions from the loop. The optimized loop is equivalent to the following:

```
R = J/L
DO 100, I=1,100
  K(I) = R + I**2
100 CONTINUE
```

The following example illustrates a form of invariant code that is a little less obvious.

```
DIMENSION B(10,10,10)
DO 10 I=1,N
10 B(I,7,K) = I
```

In this sequence, the relative locations of the elements of array B are calculated by the formula

$$I - 1 + 10 * ((6 + 10 * (K-1)))$$

The value of I is the only value in this expression that changes during execution of the loop. Thus, the following subexpression is invariant:

$$-1 + 10 * ((6 + 10 * (K - 1)))$$

The compiler recognizes the invariant expression and moves it out of the loop.

In order to remove invariant code from a loop, the compiler must be certain the code is actually invariant. If the compiler cannot make this determination, it does not remove the code. The following examples show some common situations where the compiler cannot determine whether or not a given sequence is invariant.

```
COMMON X
DO 100 I=1,N
  A(I) = X**2
  B(I) = Y/Z
  CALL MYSUB (Q,R,Z)
100 CONTINUE
```

The expressions X\*\*2 and Y/Z might be invariant. But X is in common and Z is in an argument list within the loop. Thus, the call to MYSUB might change the value of either X or Z. Therefore, neither X\*\*2 nor Y/Z can be removed from the loop.

```
LOGICAL L
DO 100 I=1,N
  IF (L) GO TO 10
  J = K + M <----- Possible invariant code
110 A(I) = B(I) + C(I)
100 CONTINUE
```

The statement J = K + M might be invariant. But because of the conditional branch, that statement might never be executed. The compiler will remove the calculation K + M from the loop, but the store into J must remain inside the loop.

You can help the compiler recognize invariant code in several ways:

- Write expressions within loops so that invariant subexpressions are easier to recognize. For example in the loop

```
DO 10 I=1,N
  A(I) = 1.0 + B(I) + X
10 CONTINUE
```

the compiler might not recognize  $1.0 + X$  as invariant. But if you rewrite the assignment statement as follows:

```
A(I) = 1.0 + X + B(I)
```

the compiler recognizes  $1.0 + X$  as invariant and moves it out of the loop.

- Recognize cases where the compiler can't determine whether a code sequence is invariant, and make that determination yourself. For example, in the sequence

```
DO 100 I=1,N
  B(I) = Y/Z
  CALL MYSUB (Q,R,Z)
100 CONTINUE
```

the compiler can't move  $Y/Z$  out of the loop because `MYSUB` might change the value of  $Z$ . But if you know that `MYSUB` will not change the value of  $Z$ , you can rewrite the loop yourself.

- Avoid using zero trip loops in your program. A zero trip loop is a loop that is executed zero times. The compiler cannot remove instructions from such a loop, because to do so might change the results of the program. Zero trip loops are discussed later in this chapter under `ONE_TRIP_DO` Parameter.

## Replacing Slower Operations With Faster Ones

During the optimization process, the compiler replaces certain slower arithmetic operations by faster ones. In particular, multiplications (which are relatively slow) are replaced by additions (which are relatively fast).

For example, multiplication by 2 can be replaced by a single addition. Thus, the compiler replaces

```
J = 2*I
```

with

```
J = I + I
```

## REPLACING SLOWER OPERATIONS

Another example of replacing a multiplication by an addition is illustrated by the following loop:

```
DO 50 I=1,100
  B(4*I+3) = 2.5
50 CONTINUE
```

Since the variable I varies linearly (it is incremented by 1 on each pass through the loop), the multiplication  $4*I$  can be replaced by an addition. Thus, the compiler generates code equivalent to

```
  J = 3
DO 50 I=1,100
  J = J + 4
  B(J) = 2.5
50 CONTINUE
```

so that an addition replaces the multiplication.

In many situations, you can do the replacement yourself. For example, in the loop

```
DO 6 I=1,1000
  J = I*5
  A(J) = 0.0
6 CONTINUE
```

you can eliminate a multiply and a store by rewriting the loop as:

```
DO 6 J=1,5000,5
  A(J) = 0.0
6 CONTINUE
```

The following example illustrates another way you can simplify the structure of loops:

```
PRINT*, (B*(11-I),I=1,10)
```

The subscript expression  $(11-I)$  in the implied DO list results in a rather complicated subscript calculation. You can simplify this calculation by using a negative indexing parameter as follows:

```
PRINT*, (B(I),I=10,1,-1)
```

The rewritten statement will execute much faster than the original statement.



## Additional Optimization Techniques

Following are some additional ways you can change your source program to make it more efficient.

### Loop Unrolling

You can reduce the amount of time required to execute a DO loop through a technique called loop unrolling. Loop unrolling reduces the overhead resulting from incrementing and testing the DO variable by reducing the number of times the loop is executed.

In loop unrolling, you replace the loop with an equivalent straight-line sequence of statements. For example:

DO Loop Before Unrolling:

```
DO 100, I=1,4
  X(I) = A(I) + B(I)
100 CONTINUE
```

After unrolling:

```
X(1) = A(1) + B(1)
X(2) = A(2) + B(2)
X(3) = A(3) + B(3)
X(4) = A(4) + B(4)
```

In this example, the loop is completely replaced by a sequence of assignment statements, and the time required to increment and test the DO variable I and branch to the top of the loop is eliminated.

Loop unrolling makes a program longer and more complicated, and is clearly not practical if the number of loop iterations is large. Sometimes, however, you can partially unroll the loop. The following example shows how to partially unroll a loop:

Before unrolling:

```
DO 100 I = 1,1000
  X(I) = Z(I)**2
100 CONTINUE
```

After partial unrolling:

```
DO 100 I=1,1000,2
  X(I) = Z(I)**2
  X(I+1) = Z(I+1)**2
100 CONTINUE
```

In the partially unrolled loop, only half as many increment, branch, and test instructions are executed.

Note that unrolling each assignment statement into two assignment statements only works if the loop is executed an even number of times.

## Combining Loops

Another method for reducing the time required for incrementing, testing, and branching in a loop is to combine loops. The following example illustrates this technique:

Before combining:

```

DO 100 I=1,K
  A(I) = B(I) + C(I)
100 CONTINUE
DO 100 I=1,K
  E(J) = F(J) + G(J)
110 CONTINUE

```

After combining:

```

DO 100 I=1,K
  A(I) = B(I) + C(I)
  E(I) = F(I) + G(I)
100 CONTINUE

```

The combined loop reduces by half the amount of overhead associated with the original loops.

Note that loops to be combined must be iterated the same number of times.

## Using Common Blocks and Equivalence Statements

You can help the compiler optimize by avoiding the unnecessary use of common blocks and equivalence statements.

If a variable is not in common, the value of the variable remains in a register when a function or subroutine is called.

But if a variable is in common, the compiler must store the variable before every function or subroutine reference, because the compiler can't determine whether that variable is used in the referenced subprogram.

The following example shows how placing a variable in common can increase execution time:

```
COMMON I, A(1000), B(1000)
DO 111 I=1,1000
  A(I) = 4.0*B(I)
  CALL SUB (C,D)
111 CONTINUE
```

Because I is in common, its value must be stored before each call to SUB in case I is referenced within SUB. If you knew that I was not referenced in SUB, you could remove I from common, thereby eliminating the unnecessary stores.

Careless use of EQUIVALENCE statements can prevent the compiler from performing certain optimizations. Following is a typical example:

```
DIMENSION X(100)
EQUIVALENCE (X(1),W)
.
.
.
W = Y
PRINT*, X(1)
END
```

Without the EQUIVALENCE statement, the compiler would remove the statement  $W = Y$  because the value of W is not referenced in the program. But because W is equivalence to X(1), and the PRINT statement might reference X(1), the assignment statement can't be eliminated.

## Avoiding Mixed Mode Arithmetic

You should avoid mixed mode arithmetic wherever possible, because each conversion from one data type to another requires additional instructions.

Exponentiation is an exception. You should use integer values for exponents regardless of the data type of the base.

If you have a choice among modes, choose according to the following hierarchy:

```
Integer (fastest)
Real
Double Precision (slowest)
```

You should avoid double precision arithmetic because it is particularly slow. Real (single precision) arithmetic provides enough precision for most programs.

## Replacing Assignment Statement With DATA Statements

You should use DATA statements, instead of assignment statements, to initialize variables, especially if a program is to be compiled and loaded only once but executed many times. Unlike assignment statements, DATA statements are processed when the program is loaded, and require no execution time.

As with most of the other optimizations, this one can significantly reduce execution time when used with DO loops. For example, the loop

```
DO 5 I=1,1000
    A(I) = 0.0
100 CONTINUE
```

can be replaced by the statement

```
DATA A/0.0*1000/
```

for a sizable reduction in execution time.

## Efficient Branching Techniques

Following are some suggestions for improving branching efficiency within a program:

- If one branch of an arithmetic IF statement immediately follows that statement, the compiler generates more efficient code, because a branch to the immediately-following statement is not required.
- Block IF statements generate the same code as arithmetic IF statements. You should use them wherever possible, though, because they greatly improve the readability of a program.
- For more than four or five branches, a computed GO TO statement is faster than IF statements. But a computed GO TO with only two or three branches should be replaced by an IF statement. Use the computed GO TO sparingly, though, because it makes a program harder to read.

## Minimizing Subroutine and Function References

Try to minimize the number of subroutine and function references. By placing the referenced code inline, you can reduce the overhead associated with passing parameters, saving registers, and branching to and from the subprogram.

But be careful! This technique makes the program less modular (and less readable) and can sometimes prevent other optimizations by making the program excessively large.

In some cases, you can substitute a statement function for an external function. Because statement functions are expanded inline, no return jump code is required.

Another way of reducing the number of external references is by consolidating references to external functions. For example, the statement

$$A = \text{ALOG}(C) + \text{ALOG}(D)$$

can be rewritten as

$$A = \text{ALOG}(C + D)$$

## Factoring Expressions

You can sometimes decrease the number of operations required to evaluate an expression by factoring the expression. For example,

$$X = A*C + B*C + A*D + B*D$$

can be rewritten as

$$X = (A + B)*(C + D)$$

The first form performs four multiplications and three additions. The second form performs one multiplication and two additions.

## FORTRAN Command Parameters

Three parameters on the FORTRAN command influence the optimizations performed by the compiler. These parameters are

The OPTIMIZATION Parameter

The EXPRESSION\_EVALUATION Parameter

The ONE\_TRIP\_DO Parameter

## OPTIMIZATION Parameter

The OPTIMIZATION parameter controls the level of optimization performed by the compiler. Options for this parameter are

- DEBUG Produces same optimization as LOW, but adds special code for use by the Debug facility.
- LOW Results in very little optimization (but faster compilation).
- HIGH Results in a highly optimized program (but slower compilation).

Thus, for a fully optimized program, you should specify OPTIMIZATION=HIGH. But remember that while you're debugging the program, you should specify OPTIMIZATION=DEBUG for faster compilation (or omit the OPTIMIZATION parameter since DEBUG is the default option).

## EXPRESSION\_EVALUATION Parameter

The EXPRESSION\_EVALUATION parameter affects the optimization of expressions. If you omit this parameter, the compiler will perform optimizations that may alter the results of the program. If you want to prevent these kinds of optimizations, you can specify any or all of the following options for the EXPRESSION\_EVALUATION parameter:

- EXPRESSION\_EVALUATION=MAINTAIN\_EXCEPTIONS
- EXPRESSION\_EVALUATION=CANONICAL
- EXPRESSION\_EVALUATION=MAINTAIN\_PRECISION

## MAINTAIN\_EXCEPTIONS Option

During the normal course of optimizing expressions, the compiler may replace "unsafe" operations by "safe" ones. (A "safe" operation is one that cannot cause an execution error. An "unsafe" operation is one that might.) The MAINTAIN\_EXCEPTIONS option prevents the compiler from performing those kinds of optimizations.

For example, if the compiler encounters the expression

```
0.0/x
```

it might optimize that expression by changing it to simply

```
0.0
```

This saves the time required to do the divide. But suppose that X had a value of zero. In the first expression, an execution error would result, while in the second expression, a value of zero for X would have no effect. The `MAINTAIN_EXCEPTIONS` option would prevent the compiler from replacing the first expression with the second one.

The decision of whether to preserve any unsafe operations in a program is up to you. Such operations can provide a useful check for incorrect data. It is sometimes better for the program to abort rather than to continue executing with incorrect values. But allowing the compiler to replace unsafe operations with safe ones generally results in a faster program.

## CANONICAL Option

When the compiler optimizes expressions, it may regroup operations within the expression in order to eliminate operations or form common subexpressions. Although the regrouping is guaranteed to be mathematically equivalent, it may not always be computationally equivalent. Thus, the result of an optimized expression may differ slightly from the result of the original expression. For example, the results of the expression

$$I + J - I$$

may be regrouped to form the expression

$$J$$

But suppose that I had an incorrect value, such as an indefinite value. In that case, the first expression would cause a runtime error whereas the second expression would execute correctly.

The `CANONICAL` option of the `EXPRESSION_EVALUATION` parameter causes expressions to be evaluated strictly according to the precedence rules for arithmetic expressions. The precedence rules are:

**\*\*** operations are evaluated first

**\*** and **/** operations are evaluated next (in the order they are encountered)

**+** and **-** operations are evaluated last (in the order they are encountered)

For example, the `CANONICAL` option would prevent the compiler from changing

$$I + J - I$$

to

$$J$$

Although the program would execute more slowly, this would ensure that the program would terminate if I had an indefinite value.

Subexpressions enclosed in parentheses are always evaluated before being combined with other subexpressions, regardless of whether you specify the CANONICAL option. Thus if you wrote

$$(I + J) - I$$

I would be subtracted from the result of I + J.

## MAINTAIN\_PRECISION Option

The MAINTAIN\_PRECISION option of the EXPRESSION\_EVALUATION parameter prevents the compiler from performing optimizations that change the form of an expression, unless the result of the changed form is exactly the same as the result of the original form.

For example, consider the expressions

$$A/3.0 \quad A*0.33333$$

These two forms are mathematically equivalent, but not computationally equivalent. That is, the result of the first form will have greater precision than the result of the second form.

Thus, for full optimization, you should omit the MAINTAIN\_PRECISION option. But remember that the results of the optimized program may differ slightly from the results of the original program.

## ONE\_TRIP\_DO Parameter

The ONE\_TRIP\_DO parameter affects the optimization of DO loops. This parameter tells the compiler that the program contains no zero trip loops (all loops are executed at least once). Zero trip loops influence optimization in the following way.

A zero trip loop results when the terminating conditions of a loop are satisfied before the loop is executed. For example, the following is a zero trip loop:

```

N = 11
DO 10 I=N,10
  A(I) = A(I) + 1.0
10 CONTINUE

```

The index variable N has a value of 11 when the DO statement is executed. Thus, the loop will not be executed.



One of the ways the compiler optimizes a loop is to remove certain instructions from the loop and place them outside the loop so that they are not repeated each time the loop is executed. For example, in the loop

```
DO 10 I=1,100
  A = 0.0
  IF (B(I) .EQ. A) GO TO 20
10 CONTINUE
```

the statement `A = 0.0` can be moved outside the loop. This makes the loop faster. (`A = 0.0` is executed only once instead of 100 times.)

But if the loop is a zero trip loop, moving `A = 0.0` outside the loop could change the results of the program. Therefore, the compiler will move statements outside a loop only if the loop will be executed at least once.

The compiler has no way of “knowing” whether or not a given loop is a zero trip loop. But you can provide this information when you call the compiler by specifying `ONE_TRIP_DO=ON` on the FORTRAN command. The compiler is then free to optimize all DO loops. If you omit this parameter, the compiler assumes that the program contains zero trip loops, and does not optimize DO loops.

For full optimization, then, you should specify `ONE_TRIP_DO=ON` on the FORTRAN command. But be sure that your program does not contain any zero trip loops. Otherwise, the results of the program could be changed (or execution errors could result).

## Summary of Optimizing a Program

Optimizing is the process of altering a program to make it execute faster. The purpose of optimizing a program is to reduce the total cost of that program. Optimizing reduces the cost of a program by reducing the amount of computer time required by the program. However, the time required to write, debug, modify, and maintain a program is also an expensive resource. Therefore, you should avoid optimizations that require much of your time, or that make a program more difficult to maintain.

The FORTRAN compiler is capable of performing many optimizations. The most important optimizations you can perform are those that help the compiler perform its own optimizations.

The compiler has the ability to recognize certain unnecessary operations and to eliminate them from the code. Some of the operations that can be eliminated are:

- Identity instructions
- Constant subexpressions
- Dead instructions
- Common subexpressions

Some of the ways you can help the compiler recognize unnecessary operations are:

- Keep program units short (less than 100 lines).
- Avoid careless programming that results in extra operations.
- Write expressions so that the compiler can recognize common subexpressions.

The most beneficial optimizations are those that remove operations from loops. When the compiler optimizes a program, it tries to remove certain instructions from loops so that the instructions are not repeated. Eliminating unnecessary instructions from loops often significantly decreases execution time.

Another type of instruction that the compiler can remove from a loop is the invariant instruction. An invariant instruction is one that can be removed from a loop without changing the results of the program.

In order for the compiler to remove instructions from a loop, it must be certain that those instructions are actually invariant. Sometimes that determination cannot be made during compilation.

You can help the compiler recognize invariant code in the following ways:

- Write expressions so that invariant subexpressions are easier to recognize.
- Recognize cases where the compiler can't determine whether or not a sequence is invariant, and remove the sequence yourself.

Another important optimization technique is the replacement of slower operations by faster ones. Because addition is faster than multiplication, the compiler attempts to replace multiplications by additions. For example,

$$J * 2$$

is replaced by

$$J + J$$

In some cases, you can do the replacement yourself. For example, you can rewrite the loop

```

DO 6 I=1,1000
  J = I*5
  A(J) = 0.0
6  CONTINUE

```

as follows:

```

DO 6 J=1,5000,5
  A(J) = 0.0
6  CONTINUE

```

Following are some additional techniques for optimizing a program:

- Expand short loops into straight-line code.
- Combine loops wherever possible.
- Avoid unnecessary use of common blocks and EQUIVALENCE statements.
- Avoid mixed mode arithmetic.
- Use DATA statements, instead of assignment statements, to initialize variables and arrays.
- Use computed GO TO statements when more than four or five branches can be taken. Use IF statements otherwise.
- Replace subroutine and function references by inline code.
- Factor expressions to reduce the number of operations needed to evaluate the expressions.

Three parameters on the FORTRAN command influence the optimizations performed by the compiler:

The OPTIMIZATION parameter controls the level of optimization. For full optimization, specify OPTIMIZATION=HIGH. But remember that this option results in slower compilation. Thus, during your debugging runs, you should omit the OPTIMIZATION parameter (default option is DEBUG).

The EXPRESSION\_EVALUATION parameter prevents the compiler from performing optimizations that could change the results of expressions. For full optimization of expressions, omit the EXPRESSION\_EVALUATION parameter.

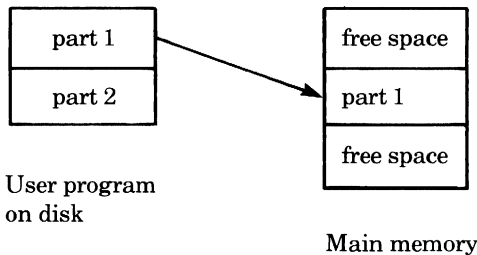
The ONE\_TRIP\_DO parameter informs the compiler that all DO loops are executed at least once. This knowledge enables the compiler to perform certain optimizations on the loops. A loop that is executed zero times cannot be optimized to the fullest extent. For maximum optimization, specify ONE\_TRIP\_DO on the FORTRAN command, but be sure that your program contains no zero trip loops.

Virtual memory is a concept that enables you to address the computer's main physical memory as though it were unlimited. In a virtual memory system, you can write programs that are larger than the amount of physical memory available. In a multiprogramming environment, such as NOS/VE, several programs can run in an area that is smaller than the total area required by all the programs. The main advantage of a virtual memory system is that it requires no knowledge or intervention on the part of the programmer. In fact, most programmers who use it are completely unaware of its existence.

A virtual memory system frees you from the necessity of using such techniques as overlays and segmentation for larger programs. With virtual memory, you can assume that you have an unlimited amount of main memory, although, as we will see later, you do have limited control over how efficiently virtual memory operates.

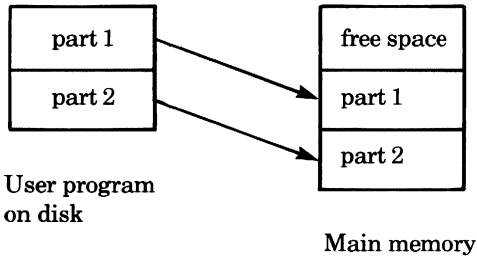
## What Is Virtual Memory?

The guiding principle behind virtual memory is that for a program to execute, the entire program does not need to be in main memory at one time. In a virtual memory system, only that portion of a program required for execution at a particular time is in main memory. The portion of the program not needed for execution is kept on external (disk) storage. The following example shows a program divided into two parts. Initially, only the first part is loaded into memory. Other areas of memory are free for other uses.

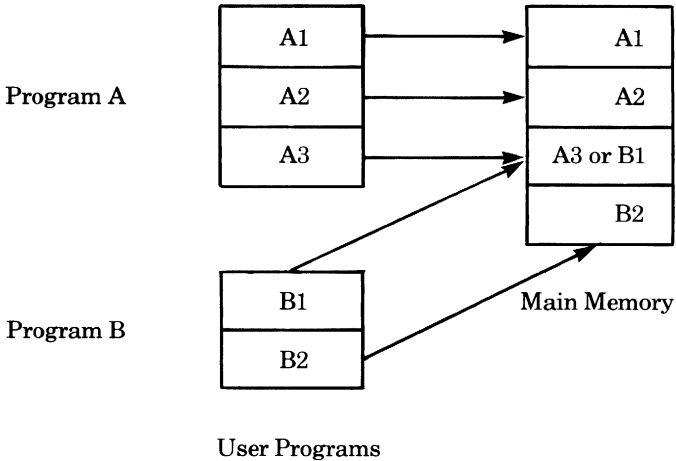


## WHAT IS VIRTUAL MEMORY?

When a portion of the program on external storage is needed for execution, it is automatically loaded into main memory:



If a vacant area of memory cannot be found, the system finds an area that is not being used, unloads (copies to disk) its contents, and loads the required information in its place. Thus, two or more programs whose combined size exceeds the amount of available main memory can execute successfully by sharing the same area of memory. The following illustration shows two programs sharing a limited amount of physical memory:



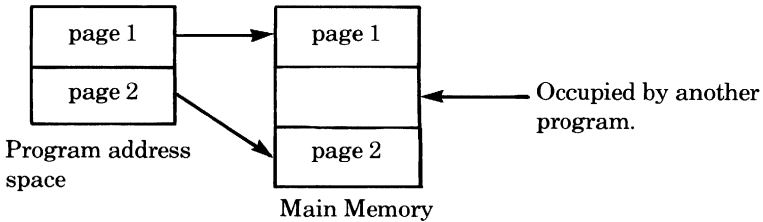
Combined, programs A and B exceed the available amount of main memory. However, by sharing a portion of main memory both programs can run together.

The management of memory and the loading and unloading of parts of programs during execution occurs without the knowledge of the programmer. Although you don't need to know anything about virtual memory in order to take advantage of its benefits, a basic understanding of the principles involved can help you write programs that use virtual memory more efficiently.

In a virtual memory system, the collection of all the addresses occupied by a program is known as the program's address space. A program's address space is divided into units of equal length called pages. The computer's main memory is divided into page-size units called page frames.

Not all of a program's address space needs to be in memory at the same time for the program to execute. Only pages required for execution are loaded at a given time. Pages not required are kept on external storage (disk). During execution, pages are loaded as needed into available page frames. When pages are no longer needed, they can be unloaded to make room for other pages.

The pages of a program's address space that are in memory at a given time need not be contiguous; that is, they can be scattered throughout memory. They are logically contiguous as far as the programmer is concerned. This freedom enables the program to make efficient use of available free space within the system. The following diagram shows a program consisting of two pages. The pages are loaded into available page frames in main memory.



Although the program's address space is not physically contiguous in memory, it appears to the programmer to be contiguous.

Because the pages of a program's address space need not be contiguous in memory, the system has a high degree of flexibility in loading and unloading pages. A page can be loaded wherever the system can find a vacant page frame.

## How Does Virtual Memory Work?

When you begin execution of a program, the system loads the first page (the page containing the main program) into memory. When the program references an address in virtual memory (that is, an address in a page that is not currently loaded), a system interrupt known as a page fault occurs: The system detects that a virtual address has been referenced, locates the referenced page, and loads it into an unused page frame in main memory.

But what happens if a page fault occurs and no vacant page frames are available? In this case, the system copies an existing page onto external storage and loads the required page in its place. This is known as a page swap. But how does the system know what page to swap out?

The system determines which page in memory to swap according to a process known as a page replacement algorithm. The page replacement algorithm is based on the theory that if a page has not been referenced for a long time, it probably will not be referenced in the near future; that page can therefore be safely removed. Therefore, the page that is swapped out is the page that has not been referenced for the longest period of time. Because numerous page swaps can be time-consuming, an effective page replacement algorithm should attempt to minimize the number of page swaps that occur.

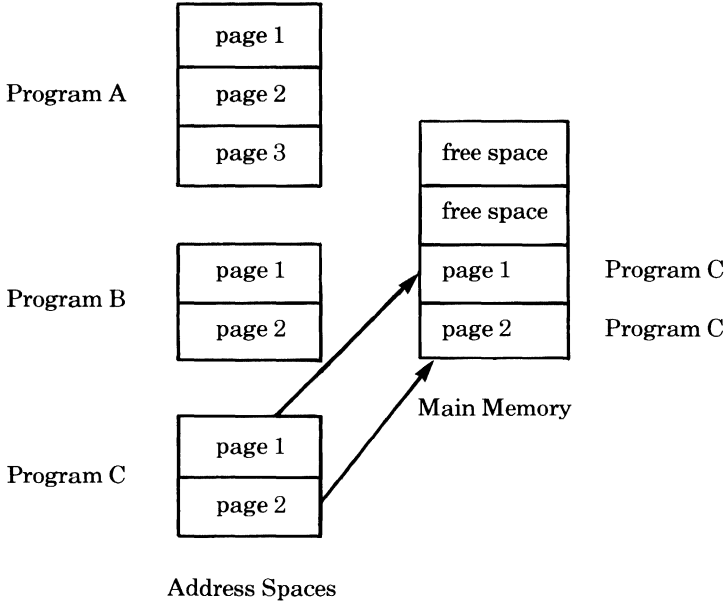
When an executing program references an address outside of the current page, the system must first determine whether the referenced page is in memory.

If the referenced page is in memory, the system uses internally-maintained tables to locate the referenced address.

If the referenced page is not in memory, a page fault occurs and the system loads the new page into memory. If the system cannot find a vacant page frame, a page swap occurs. Using the page replacement algorithm, the system determines which page currently in memory can be replaced, copies that page to disk, and loads the new page in its place.

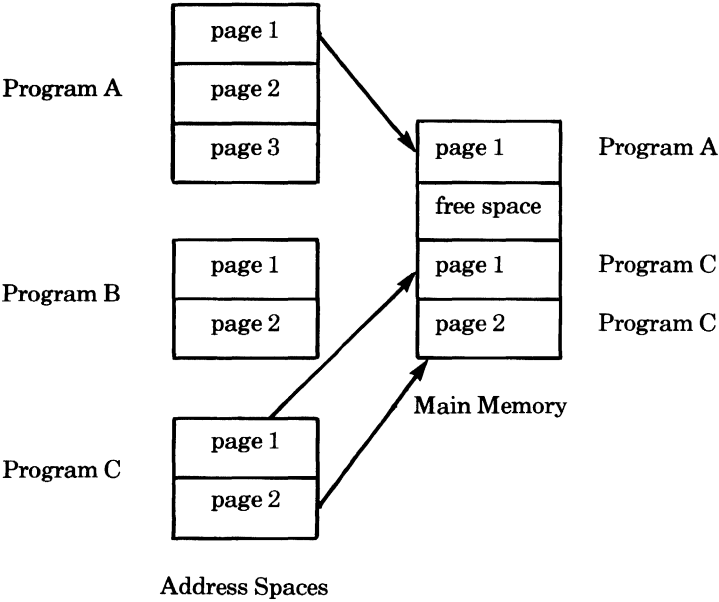


The following diagrams show an example of three programs sharing a limited amount of main memory. The diagrams show the memory layout at four different times during the programs' execution. The first diagram shows the memory layout at time t1. Program C is executing in main memory. Program C occupies two page frames, leaving two empty page frames.

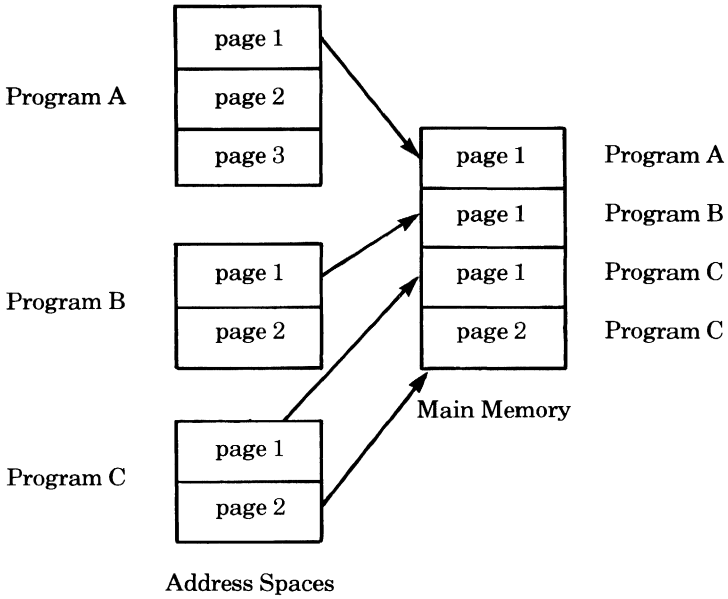


HOW DOES VIRTUAL MEMORY WORK?

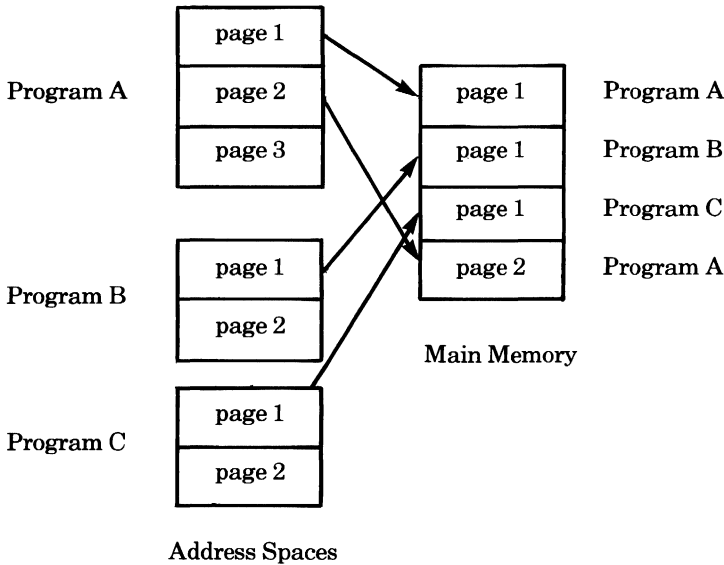
At time t2, program A begins executing. Its first page is loaded into an empty page frame in main memory. The following diagram shows the memory layout at time t2:



At time t3, program B begins executing. Its first page is loaded into the last available page frame in main memory. The following diagram shows the memory layout at time t3:



At time  $t_4$ , program A references an address in page two. A page swap occurs: Page 2 of program C is copied to disk and page 2 of program A is loaded in its place. The following diagram shows the memory layout at time  $t_4$ :



## Programming Guidelines

Virtual memory allows you to write programs as though you had an unlimited amount of main memory available. However, the overhead required to load and unload pages as a program executes can be time consuming. Under certain conditions, the amount of page swapping can become excessive, and a phenomenon known as page thrashing results. Page thrashing is a condition where pages are unloaded and immediately loaded again. Page thrashing is extremely wasteful of system resources. Your goal should be to reduce the amount of paging that occurs.

You, the programmer, have little direct control over the paging operations of the operating system. The allocation of data and code to pages and the determination of which pages to load and unload are all under control of the system. The page replacement algorithm, which determines what pages are to be replaced, is designed to minimize page thrashing. However, program design also plays an important role in efficient paging operations.

Although virtual memory provides the appearance of an unlimited amount of main memory, the actual amount is limited. A smaller program requires fewer pages and, therefore, is likely to require fewer paging operations. You should therefore continue to apply the same programming techniques for conserving memory that you applied in a nonvirtual system. Some general guidelines for efficient use of virtual memory are as follows:

Concentrate your efforts on the larger, slower program units.

Avoid techniques that are system-dependent.

Compile under `OPTIMIZATION=HIGH`, since that generates less code.

Use a structured style of programming that reduces branching.  
(Structured programming is discussed in chapter 2, Debugging.)

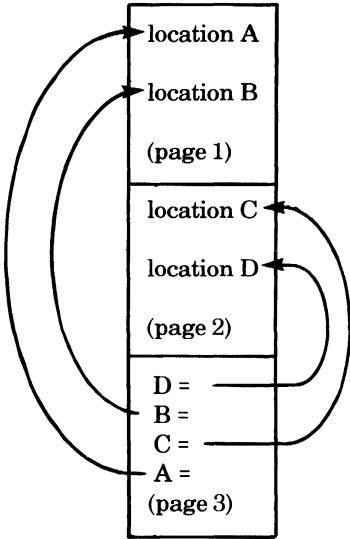
## The Locality of Reference Concept

Before beginning a discussion of efficient use of virtual memory, you should understand a concept known as "locality of reference". Locality of reference is a measure of how close together the memory references within a program are. Basically, locality of reference means keeping address space references within a program as close together as possible for as long as possible.

Programs with a high degree of locality of reference tend to be more efficient in a virtual memory system.

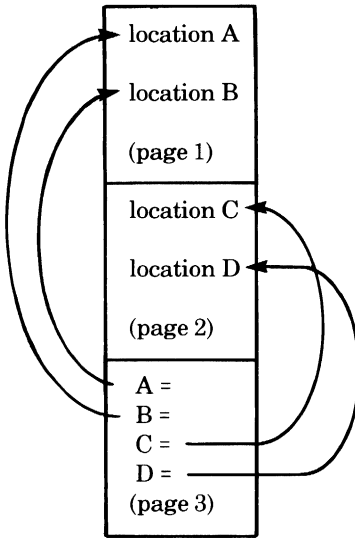
Locality of reference has both a time-related aspect and a distance-related aspect. That is, a program whose references to the same, or nearby, locations are separated by a relatively long span of time is said to have poor locality of reference. And a program that references widely separated memory locations within a relatively short span of time is said to have poor locality of reference.

The following diagram shows an example of a program with poor locality of reference:



In this program, pages 1 and 2 contain data and page 3 contains instructions. Page 3 references page 2, then page 1, then page 2 again. These widely scattered address space references may cause unnecessary paging.

The following diagram shows the same program reorganized for better locality of reference. Now, two references to page 1 are followed by two references to page two.



Why do programs with a high degree of locality of reference tend to be more efficient than programs without this quality?

Earlier in this chapter we discussed the page replacement algorithm that the system uses to decide which pages in memory to replace when a page fault occurs and there are no vacant page frames in memory. This algorithm is based on the following generalizations:

Once a location is referenced, it will probably be referenced again soon. For example, DO loops often reference the same location numerous times.

Once a location is referenced, a nearby location will probably be referenced shortly thereafter. For example, operations on arrays reference locations that are close to one another.

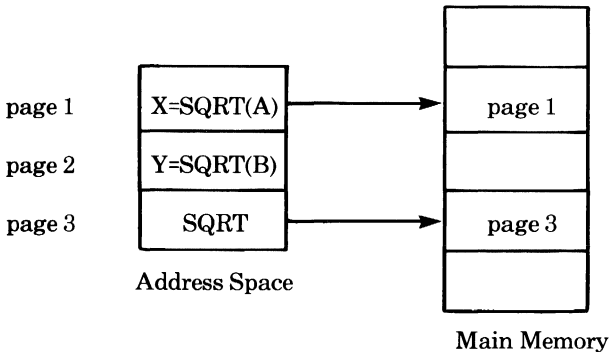
When it becomes necessary for the system to replace a page in memory, the system determines which pages contain addresses that have not been referenced for the longest period of time. Those pages are candidates for replacement.

If the memory references within a program are widely scattered throughout a program's address space, there is an increased chance that the referenced locations will be in pages that are not currently loaded. And if references to the same, or nearby, locations are separated by a long period of time, there is an increased likelihood that the pages containing the referenced locations will be unloaded between references.

Although you have no way of knowing exactly how your program is allocated among pages, and what pages are loaded at a given time, you can improve paging efficiency by maximizing the locality of reference in your program.

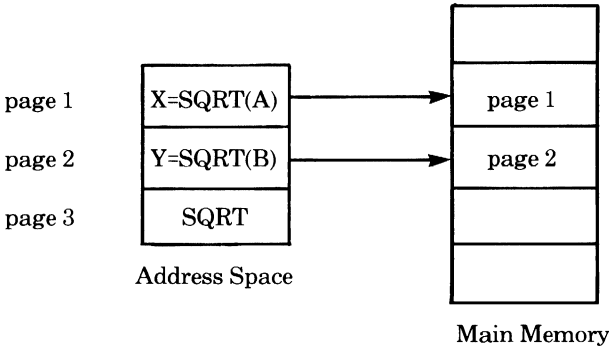
The following example shows how program structure can affect locality of reference.

Suppose we have a fairly long program that consists of three pages. The program contains two calls to the SQRT function that are widely separated (the first call occurs in the first page, and the second call occurs in the second page). The program has poor locality of reference. Assume that the SQRT function is located in the third page. When we begin execution of the program, the first page is loaded into memory. When SQRT is called, page three is also loaded. The following diagram shows the memory layout at the time SQRT is loaded:

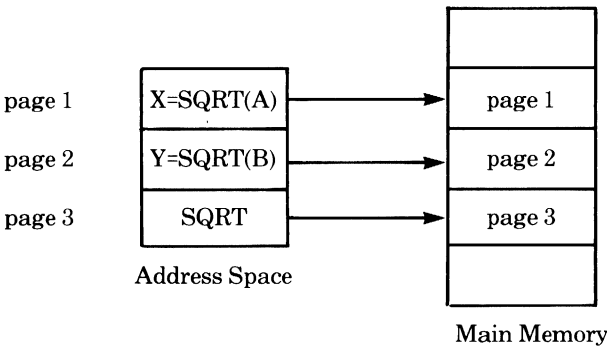




Page 1 (the page containing the main program) will remain in memory throughout execution. But page 3 is a candidate for removal. The longer page 3 remains in memory without being referenced, the greater the likelihood that it will be swapped out to make room for another page. Chances are good that by the time page 2 is loaded, page 3 will no longer be in memory, as shown in the following diagram:



When  $Y=SQRT(B)$  is executed, a page fault occurs and page 3 is reloaded:



If we could improve the locality of reference by rearranging the program so that the two calls to SQRT were closer together, there would be a much greater probability that page 3 would need to be loaded only once.

Related to the idea of locality of reference is the concept of the working set. A program's working set is the minimum number of pages that must be in memory for the program to run efficiently. Pages that are referenced frequently tend to remain in memory as part of the working set, whereas pages that are not referenced frequently tend to remain on external storage.

A program with good locality of reference will tend to have a more stable working set, that is, the working set can be maintained with fewer paging operations. A program with poor locality will have a working set that is constantly changing as required pages are swapped.

## Suggestions for Improving Locality of Reference

Most of the guidelines for programming in virtual memory are aimed at improving the degree of locality of reference in a program. A general rule for improving locality of reference is: "Keep memory references as close together as possible for as long as possible", because the closer together (both in time and in space) those references are, the less the chance of causing page faults.

Following are some suggestions for improving locality of reference:

1. Wherever possible, organize your program so that all references to a particular data area are close together. For example, you should initialize variables and arrays immediately before using them, rather than at the beginning of the program. This helps keep the references to the variables and arrays closer together, thus improving locality.
2. Reference data in the order it is stored, or store data in the order it is referenced. By referencing locations that are close together in memory you can reduce the possibility of page faults. For example, recall that in FORTRAN, arrays are stored in columnwise order. You should complete the references to a single column before referencing elements in the next column.

The following example shows how an array A(512,10) would be stored in memory, assuming a page length of 512 words. The example shows two DO loops that initialize array A.

page 1	A(1,1) . . . A(512,1)
page 2	A(1,2) . . . A(512,2)
	.
	.
	.
page 10	A(1,10) . . . A(512,10)

Loop to initialize array A in rowwise order:

```

DO 5 I=1,512
  DO 5 J=1,10
5   A(I,J) = 0.0

```

Loop to initialize array A in columnwise order:

```

      DO 5 J=1,10
      DO 5 I=1,512
5     A(I,J) = 0.0

```

The first loop initializes array A in rowwise order, beginning with A(1,1), followed by A(1,2), followed by A(1,3), and so forth. Notice that each execution of statement 5 references a different page. By changing the loop to reference array A in columnwise order, the references within a page are completed before the next page is referenced.

3. Try to reduce the number of external calls in your program, because they often cause references to distant locations. (Note that this guideline conflicts with structured programming rules, which encourage multiple modules.) Ways of reducing external calls include:

- Place shorter subroutines and functions inline.
- Wherever possible, use statement functions instead of external functions.
- Avoid implied DO lists in input/output statements, because (in certain cases) each iteration of the list generates an external reference. For reading to or writing from an array, you can substitute the array name for an implied DO list. For example, if A, B, and C are 100-element arrays, you can replace

```
READ (2, 100) (A(I), B(I), C(I) I=1,100)
```

with

```
READ (2, 100) A, B, C
```

4. If a program has nested calls, store the subroutines in the same order they are called. For example, if subroutine A calls subroutine B, and subroutine B calls subroutine C, the pages containing those subroutines must all be in memory concurrently. If A, B, and C are stored in sequence, fewer pages may be required.
5. If your program makes repeated calls to routines such as SQRT, try to group those calls together whenever possible. If the calls are close together, the page containing the routine will probably remain in memory. If the calls are widely separated, the page may have to be reloaded.

6. Store data used only by specific routines along with the routine, if possible. Avoid using COMMON for data that is local to a particular subroutine. Putting data in COMMON can separate it from the routine, resulting in poor locality.
7. Group high-use data areas and subroutines together if possible. This can result in a more stable working set. The pages containing the frequently used code will tend to remain in memory, whereas the pages containing seldom-used areas will tend to remain on external storage. Mixing high-use and low-use areas can result in frequent paging operations.
8. Try to use COMMON as efficiently as possible. For example, if possible, use the same area for different data in different phases of a program. (But be careful, as this can lead to programming errors.)
9. Store data as close as possible to other data used concurrently. For example, if a program operates on two arrays in the same DO loop, store the arrays next to each other in a common block for better locality of reference.
10. Replace arithmetic and logical IF statements with block IF structures. A program that flows from top to bottom will tend to have better locality than one that branches extensively.
11. Use methods for reducing the storage required by sparse arrays. A sparse array is one in which most of the elements are zero. You can develop methods for storing such arrays that eliminate the storage needed for the zero-valued elements.

For example, consider the following elements in an array A(6,6):

```
0 0 2 0 0 0
5 0 0 3 0 0
0 0 0 0 0 4
0 6 0 0 0 0
0 0 0 2 0 0
0 0 1 0 0 5
```

A total of 36 words is required for this array, although only eight words contain nonzero values.

An alternate method of storing a sparse array is to first define three one-dimensional arrays. You then store the nonzero elements in one of the arrays, and the row and column positions of those elements in the corresponding positions of the other two arrays. Any element whose value is not in the first array has the value 0. Thus, the preceding array could be stored as follows:

Contents of array containing nonzero values:

2 5 3 4 6 2 1 5

Contents of array containing row position of nonzero values:

1 2 2 3 4 5 6 6

Contents of array containing column position of nonzero values:

3 1 4 6 2 4 3 6

The value of A(1,3) is 2, the value of A(2,1) is 5, and so forth. Only 24 words are required for the array instead of the original 36.

The following program shows a method for accessing an element of an array stored as described above. The nonzero elements are in array A. The arrays ROW and COLUMN contain the row position and column position, respectively, of the nonzero elements. The program puts the value of element I,J in the variable VALUE.

```

VALUE = 0
INTEGER A(8), COLUMN(8), ROW(8), VALUE
DO 12 K=1,8
  IF (ROW(K) .EQ. I .AND. COLUMN(K) .EQ. J) THEN
    VALUE = A(K)
    GO TO 15
  ENDIF
12 CONTINUE
15 CONTINUE

```

## Summary of Using Virtual Memory

A virtual memory system enables you to address the computer's main memory as though it were unlimited. Under virtual memory, you can run programs that exceed the amount of main memory available. Virtual memory eliminates the need for such methods as overlays and segmentation for larger programs.

The principle behind virtual memory is that an entire program need not be in memory at one time in order for that program to run. Portions of the program that are not being used can remain on disk, while the required portions are in memory.

In a virtual memory system, every program is divided into equal-length units called pages. When a program executes, the required pages are loaded into memory while the unneeded pages remain on disk. As execution continues, pages are loaded into memory as they are required. If necessary, pages in memory that are not being used can be unloaded (“swapped out”) to make room for required pages.

Although virtual memory operates without any knowledge or intervention on your part, the design of your program can influence how efficiently virtual memory works. In a poorly designed program, page thrashing can degrade the efficiency of the program. Page thrashing is a condition where pages are unloaded and then immediately loaded again. You can apply programming techniques that help ensure efficient paging operations.

Probably the most important aspect of program design is a quality known as locality of reference. Locality of reference is a measure of how close together the memory references within a program are. Programs with a high degree of locality of reference are likely to require fewer paging operations than programs whose memory references are widely scattered throughout the program’s area of memory.

Some suggestions for improving the locality of reference of your program are:

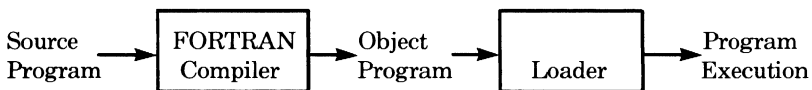
- Continue to apply techniques for reducing the size of your program. Programs that require fewer pages may require fewer paging operations.
- Organize the program so that all references to a particular area are close together.
- Reference data in the order it is stored, or store data in the order it is referenced.
- Reduce the number of external calls in the program.
- Store subroutines in the same order they are called.
- If the program makes repeated calls to a routine, try to group those calls together.
- Avoid using COMMON for data that is local to a particular routine.
- Store data as close as possible to other data used concurrently.
- Replace arithmetic and logical IF statements with block IF structures to reduce branching.

An object library is a specially-formatted file of modules. This discussion concentrates on modules that are compiled programs, although object libraries can contain other types of modules.

To understand object libraries, it is helpful understand the loading process. Therefore, this chapter begins with a discussion of the loading process and some of the most useful features of the loader. The discussion presents only the most commonly used commands and parameters used to load programs and manage object libraries. For more information on these topics, refer to the SCL Object Code Management Usage manual.

## The Loading Process

Before a compiled object program can be executed, it must be processed by a system utility called the loader. The sequence of steps leading to program execution can be diagrammed as follows:



## What the Loader Does

The loader reads the object code produced by the compiler, performs the necessary processing, and begins execution of the program.

In preparing the object program for execution, the loader performs the following sequence of steps:

1. Assigns the program to an area of memory and loads the program into that area.
2. Loads all of the routines called by the program into the assigned area.
3. Provides the program with the addresses of the called routines.
4. Begins execution of the loaded program.

## THE LOADING PROCESS

You are probably familiar with the command LGO, commonly used to execute a compiled program. Actually, LGO is a form of the name call command, which calls the loader and begins the loading process. We now take a closer look at how the loader performs the above steps.

We begin by considering the following simple main program and subroutines:

```
PROGRAM SIMPLE
READ (*, *) X
CALL SUBA (X,Y)
CALL SUBB (Y,Z)
PRINT *, Z
END

SUBROUTINE SUBA (X, Y)
Y = X + 1.0
RETURN
END

SUBROUTINE SUBB (Y, Z)
Z = Y*10.0
RETURN
END
```

This program does the following:

Reads a number.

Calls a subroutine that adds 1.0 to the number and returns the result.

Calls another subroutine that multiplies the result by 10.

Prints the final result.

Assume that program SIMPLE and subroutines SUBA and SUBB all reside on a file named PROG\_FILE. We can compile the preceding program with the command

```
FORTRAN I=PROG_FILE
```

The compiler writes the compiled object code to the default file LGO. To execute the program, we simply specify the command LGO. However, as we will show, this command does much more than begin execution of the program.



## Name Call Loading

The LGO command is a form of the name call command. A name call command consists of the name of the file containing the compiled object code. This command calls the loader and begins loader processing. The loader has various processing options that you can request through other system commands. These options are discussed later in this chapter.

The first step the loader performs is to load the program into memory. When you request a name call load by specifying an object file name such as LGO, the loader loads all the program units on that file into memory. (In loader terminology, a program unit is called a module. We use that term from throughout the rest of this chapter.)

Thus, in our sample program, we specify the command LGO. The main program SIMPLE and subroutines (or modules) SUBA and SUBB are loaded into memory.

The next step the loader must perform is to supply the program with the machine addresses of all the routines called by the program. (The compiler cannot do this because machine addresses are not known at compile time.)

Our sample program makes explicit calls to two subroutines: SUBA and SUBB. Since these subroutines have been loaded into memory, the loader can readily provide the calling program with their machine addresses.

A reference in a program unit to an entry point in another program unit is called an external reference. The process of supplying a calling program with the addresses of the called entry points is known as satisfying external references.

When the loader encounters an external reference, it attempts to satisfy that reference by searching for the entry point in modules already loaded. In our example, the calls to SUBA and SUBB are satisfied in this way. However, if the required entry points are not contained in modules already loaded, the loader must search elsewhere for those entry points. The process of searching for entry points is discussed later in this chapter.

After all the external references in the program are satisfied, the loader can then perform the last step of the loading process, which is to begin execution of the program.

Before we find out where the loader searches for entry points to satisfy external references, let's examine some additional loader capabilities.

## Loading Modules From Separate Files

We now take another look at the program discussed earlier:

```
PROGRAM SIMPLE
  READ (*, *) X
  CALL SUBA (X, Y)
  CALL SUBB (Y, Z)
  PRINT *, Z
END

SUBROUTINE SUBA (X, Y)
  Y = X + 1.0
  RETURN
END

SUBROUTINE SUBB (Y, Z)
  Z = Y*10.0
  RETURN
END
```

In the original example, the main program and subroutines were compiled together and written to file LGO. Now, however, assume that subroutine SUBB was written by a different programmer and stored on a separate file. The main program and SUBA are still stored on file LGO, but subroutine SUBB is now on file SUB\_FILE.

In this case, if we try to load and execute the program with the LGO command, the loader will not know where to find SUBB. This causes a loader error. However, NOS/VE provides a method of loading modules from separate files.

You can load modules from separate files by substituting an EXECUTE\_TASK command for the name call command. The EXECUTE\_TASK command performs the same operation as the name call command, but it provides you with some additional options. (These options are discussed later in this chapter.) We can request the loader to load subroutine SUBB from file SUB\_FILE as follows:

```
EXECUTE_TASK FILE=(LGO, SUB_FILE)
```

This command loads all of the program units on files LGO and SUB\_FILE into memory, satisfies all external references in the loaded program, and begins execution.

You can specify any number of files in an EXECUTE\_TASK command. Execution always begins with the main program regardless of which file contains the main program. For example, the command

```
EXECUTE_TASK FILE=(BIN, AFILE, DATA1)
```

loads program units from files BIN, AFILE, AND DATA1 and begins execution of the main program.

Note that when you specify a file in an EXECUTE\_TASK command, all of the modules on that file are loaded regardless of whether or not they are actually called anywhere in the program.

## Generating a Load Map

A load map is a printable output listing, produced by the loader, that shows how the loader allocated memory during the load.

The load map lists the names of all modules that were loaded and, for each module, lists

- The location in memory where the module was loaded

- The file or library from which the module was loaded

- The length (number of words) of the module

The map also lists additional information about the internal storage attributes of the modules.

Although the load map contains little information that is useful to most programmers, it can be useful as a debugging tool if other debugging methods have failed to locate the error. To use a load map, you should have a memory dump and an object listing of your program. You should also know how to interpret those listings.

You control the generation of a load map through parameters on the EXECUTE\_TASK and SET\_PROGRAM\_ATTRIBUTE commands. But whether or not you need to specify one of these commands depends on which default load map options are in effect at your site. (You can quickly find out the default load map setting by entering the command DISPLAY\_PROGRAM\_ATTRIBUTES. This will tell you, among other things, which load map options are effective and the name of the file to which the map is written.)

You can use the EXECUTE\_TASK or SET\_PROGRAM\_ATTRIBUTE command to alter any of the default load map options.

The EXECUTE\_TASK command requests a map for a single load operation. For example, in the sequence

```
EXECUTE_TASK FILE=BINA LOAD_MAP=MAPFIL LOAD_MAP_OPTIONS=(B,EP,CR)
BINB
```

the EXECUTE\_TASK command does the following:

Loads and executes modules on file BINA

Generates a load map with the B (block), EP (entry points), and CR (entry points cross reference) options

Writes the load map to a file named MAPFIL

The name call command BINB loads and executes the program on file BINB, but does not generate a load map (assuming the default condition is not to produce a load map).

The SET\_PROGRAM\_ATTRIBUTE command requests a map for all subsequent loads, until you either end the terminal session or turn the load map option off. For example, in the following sequence, a load map is generated for two name call loads:

```
SET_PROGRAM_ATTRIBUTE LOAD_MAP=MAPFIL LOAD_MAP_OPTIONS=(B,EP,CR)
BINA
BINB
```

In addition to specifying files that contain modules to be loaded and generating a load map, the EXECUTE\_TASK and SET\_PROGRAM\_ATTRIBUTE commands provide numerous other options. These commands are described in detail in the SCL System Interface manual and the SCL Object Code Management Usage manual.

When you specify a file in an EXECUTE\_TASK command, the loader loads all program units from that file regardless of whether or not they are required for execution. This can waste memory space. In our earlier example, we saw how we could load a subprogram if it resided on a different file from the calling program. The command to load and execute the program was:

```
EXECUTE_TASK (LGO, SUB_FILE)
```

where file LGO contained the main program and a subroutine named SUBA, and file SUB\_FILE contained a subroutine named SUBB.

Now, suppose that file `SUB_FILE` contained not only `SUBB` but several other subroutines as well. The preceding `EXECUTE_TASK` command would load not only `SUBB` but all of the other subroutines even though they are not called in the program.

The following discussion explains how object libraries can provide a way of avoiding the loading of unneeded modules.

## What Are Object Libraries and Why Are They Useful?

An object library is a file containing object programs in a special format that allows rapid searching and loading by the loader. The library contains a directory, so that when a particular program unit is required, the loader can go directly to that program unit without sequentially searching the library.

The advantages of loading modules from a library instead of the sequential file produced by the compiler include:

- The loader automatically searches the libraries that are available to a program and loads the required modules. With files, you must explicitly specify which files are to be used in the loading process.
- With libraries, the loader loads only those modules that are required by the program, regardless of how many modules are stored in the library. When you specify a file to be loaded, the loader loads all modules on the file.

Later, we show how you can create and modify object libraries. First, we examine how the loader uses libraries.

# How the Loader Uses Object Libraries

Consider following program:

```
PROGRAM SIMPLE
READ (*, *) X
CALL SUBA (X,Y)
CALL SUBB (Y,Z)
PRINT *, Z
END

SUBROUTINE SUBA (X, Y)
Y = Z + 1.0
RETURN
END

SUBROUTINE SUBB (Y, Z)
Z = Y*10.0
RETURN
END
```

We have seen how we can load and execute this program when the main program and subroutines are on a single file and when they are on separate files. Now, suppose that subroutine SUBB is on a library named SUB\_LIB. We can load and execute the program with the following EXECUTE\_TASK command:

```
EXECUTE_TASK FILE=LGO LIBRARY=SUB_LIB
```

The LIBRARY parameter makes library SUB\_LIB available to the loader. The loader first loads SIMPLE and SUBA from file LGO. Since the loader cannot find SUBB among any of the loaded modules, it then searches library SUB\_LIB for SUBB. Upon finding SUBB, the loader loads it into memory, satisfies the external reference, and begins execution.

Note that regardless of how many modules reside on library SUB\_LIB, only SUBB is loaded because that is the only one required by the program.

## Making Libraries Available to the Loader

We now show how you specify which libraries the loader is to search, in what order the loader searches the libraries, and how you create your own libraries. Before discussing how you make libraries available to the loader, we present a quick review of the loading process. When loading any program, the loader performs the following sequence of steps:

1. Loads all modules from the file or files specified in the name call or EXECUTE\_TASK command.
2. Satisfies all external references. The loader first searches modules already loaded for the required entry points. It then searches the available libraries. If programs loaded from libraries contain additional external references, the loader satisfies them in the same way.
3. Begins execution of the loaded program.

Before the loader will search a library, that library must be available to the loader. When the loader satisfies external references, it searches the available libraries for modules containing referenced entry points.

You can make libraries available to the loader in three ways. The first way is to declare the library in the LIBRARY parameter of an EXECUTE\_TASK command. The general form of this parameter is

**EXECUTE\_TASK FILE=file-list LIBRARY=library-list**

where library-list is a list of one or more libraries. This command adds the specified libraries to a list called the local library list.

The libraries in the local library list are available only for the EXECUTE\_TASK command in which the list appears. For example, in the sequence

```
EXECUTE_TASK FILE=ABIN LIBRARY=(LIBA, LIBB)
ABIN
```

both commands load and begin execution of the program on file ABIN. However, in the first load operation, libraries LIBA and LIBB are searched to satisfy external references. In the second load operation, LIBA and LIBB are not searched.

The second way of declaring libraries to be available for loader searching is to declare them in the ADD\_LIBRARY parameter of a SET\_PROGRAM\_ATTRIBUTE command. The general form of this parameter is

**SET\_PROGRAM\_ATTRIBUTE ADD\_LIBRARY=library-list**

This command adds the specified libraries to a list called the job library list. Libraries in the job library list are available to all subsequent load operations, until you either end the terminal session or remove the libraries from the list. For example, in the sequence

```
SET_PROGRAM_ATTRIBUTE LIBRARY=(ALIB, BLIB)
BIN1
BIN2
```

the `SET_PROGRAM_ATTRIBUTE` command makes libraries `ALIB` and `BLIB` available to the loader. Those libraries are then searched to satisfy external references in the name call loads initiated by the commands `BIN1` and `BIN2`.

You can remove one or more libraries from the job library list through the `DELETE_LIBRARY` parameter on the `SET_PROGRAM_ATTRIBUTE` command. The form of this parameter is

```
SET_PROGRAM_LIBRARY DELETE_LIBRARY=library-list
```

You can use the `DELETE_LIBRARY` parameter to remove from the local library list any libraries which you added earlier in the terminal session.

The third way of making libraries available to the loader is to create a program description module that specifies a list of libraries to be searched. You can execute a program by specifying its program description module in a name call or `EXECUTE_TASK` command. Refer to the SCL Object Code Management manual for more information about program description modules.

The following example adds two libraries to the job library list, executes two programs, then removes the libraries:

```
SET_PROGRAM_ATTRIBUTES ADD_LIBRARY=(ALIB, BLIB)
LG01
EXECUTE_TASK FILE=LG02 LIBRARY=CLIB
SET_PROGRAM_ATTRIBUTE DELETE_LIBRARY=(ALIB, BLIB)
LG03
```

When the program on file `LG01` is loaded, the loader searches libraries `ALIB` and `BLIB` to satisfy external references. When the program on `LG02` is loaded, the loader searches libraries `ALIB`, `BLIB`, and `CLIB`. (Later, we'll discuss the order in which libraries are searched.) When the program on `LG03` is loaded, the loader does not search any of the libraries `ALIB`, `BLIB`, or `CLIB`.



In addition to libraries you specify in EXECUTE\_TASK and SET\_PROGRAM\_ATTRIBUTE commands, various other libraries are available for loader searching. These default libraries are available to all programs, and need not be declared in an EXECUTE\_TASK or SET\_PROGRAM\_ATTRIBUTE command.

The default libraries are used to satisfy many of the implicit external references in a program. An implicit external reference is one that is generated by an executable statement other than a CALL statement or function reference. For example, input/output statements such as READ and PRINT generate external references to internal FORTRAN input/output routines.

One type of default library is provided in the job library list. These libraries are available in addition to the ones you add to the list via a SET\_PROGRAM\_ATTRIBUTE command. These default job libraries generally contain system-related routines that are used by most programs. An example of a default program library is the math library, which contains routines such as SIN and SQRT. (Any libraries you add to the job library list are added BEFORE the default libraries. Thus, when the loader satisfies external references, the libraries you have added will be searched first.)

A second type of default library is specified in the generated object code by the FORTRAN compiler itself. These are libraries that are required by most FORTRAN programs. An example of a compiler-specified library is the FORTRAN runtime library, which contains routines required for input/output and other operations. Compiler-specified libraries are always available when the loader loads programs written in the particular language.

A third type of library is called the NOS/VE task services library. This library contains system routines required by most programs, and is always available to the loader.

A fourth type of library resides on the job debug library list. These libraries are used by the Debug facility, and are available only if you have turned on debug mode.

We have discussed two ways of making libraries available for loader searching: by declaring them in an EXECUTE\_TASK command and by declaring them in a SET\_PROGRAM\_ATTRIBUTE command. (A third way, specifying the libraries in a program description module, is a more advanced technique and is not discussed in this manual.)

## Library Search Order

The loader always searches the available libraries in a predefined order. In most cases, the library search order will probably not be of concern to you. However, in cases where two or more libraries contain duplicate entry point names, you can determine which entry point will be loaded if you know the library search order.

When the loader satisfies external references, it first searches modules already loaded for referenced entry points. If unsatisfied external references remain after this search, the loader then searches the available libraries. Any additional external references that result from the loading of modules from libraries are satisfied in the same way.

Before the loader begins the library search, it constructs a list of libraries called the program library list. The loader then searches the libraries in order of their occurrence in this list.

The order of libraries in the program library list is as follows:

1. Local library list (as specified by the EXECUTE\_TASK command).
2. Object libraries specified by the compiler.
3. Job library list (as specified by the SET\_PROGRAM\_ATTRIBUTE command).
4. Job debug library list.
5. NOS/VE task services library.

Within each of the above lists, the loader searches the libraries in the order of their occurrence in the list.

Following are some examples of library search order. These examples assume that debug mode is off and that the job library list is initially empty.

<u>Commands</u>	<u>Order of Library Search</u>
LGO	FORTRAN-supplied libraries task services library
EXECUTE_TASK FILE=LGO LIBRARY=(ALIB, BLIB)	ALIB BLIB FORTRAN-supplied libraries task services library
SET_PROGRAM_ATTRIBUTE ADD_LIBRARY=(CLIB) EXECUTE_TASK FILE=LGO LIBRARY=(ALIB, BLIB)	ALIB BLIB FORTRAN-supplied libraries CLIB task services library

In the following example, a program library list is established and two programs are loaded and executed. The first program is loaded and executed by an EXECUTE\_TASK command that specifies two libraries in the local library list. The second program does not have a local library list.

```
SET_PROGRAM_ATTRIBUTE ADD_LIBRARY=CLIB
EXECUTE_TASK FILE=LGO1 LIBRARY=(ALIB, BLIB)
LGO2
```

When LGO1 is loaded, the following libraries are searched: ALIB, BLIB, FORTRAN-supplied libraries, CLIB, and the task services library. When LGO2 is loaded, the following libraries are searched: FORTRAN-supplied libraries, CLIB, and the task services library.

# Creating and Modifying Object Libraries

We now discuss commands, parameters, and techniques for creating and modifying object libraries.

## The CREATE\_OBJECT\_LIBRARY Utility

You create a new library, or alter an existing one, by using a system utility program called CREATE\_OBJECT\_LIBRARY. The CREATE\_OBJECT\_LIBRARY utility enables you to create new libraries, and to add, delete, or replace modules in existing libraries. The modules you place in an object library can be object modules produced by a compiler, modules from other libraries, SCL procedures, and program description modules.

To use CREATE\_OBJECT\_LIBRARY, you conduct a CREATE\_OBJECT\_LIBRARY session. A CREATE\_OBJECT\_LIBRARY session consists of the following sequence of steps:

1. Begin the CREATE\_OBJECT\_LIBRARY session.
2. Enter commands to add, delete, or replace modules in the library.
3. Generate the library.
4. End the CREATE\_OBJECT\_LIBRARY session.

To begin a CREATE\_OBJECT\_LIBRARY session, enter the command

```
CREATE_OBJECT_LIBRARY
```

The system responds with the prompt

```
COL/
```

This prompt signifies that CREATE\_OBJECT\_LIBRARY is waiting for you to input a command. After you enter a command and press the RETURN key, CREATE\_OBJECT\_LIBRARY issues another COL/ prompt and waits for you to enter another command. The session continues in this way until you enter QUIT. This ends the CREATE\_OBJECT\_LIBRARY session and returns control to the system. While you are in a CREATE\_OBJECT\_LIBRARY session, you can enter as many commands as you like. These commands tell CREATE\_OBJECT\_LIBRARY which modules are to be included in the library, which modules are to be replaced, and which modules are to be removed.

The commands you enter to add, replace, or delete modules do not actually alter the contents of a library. Instead, `CREATE_OBJECT_LIBRARY` maintains an internal list called the module list. The commands you enter add, replace, and delete libraries from this list. In order to actually create a new library, or modify an existing one, you must enter the command

**`GENERATE_LIBRARY LIBRARY=file-name`**

This command creates an object library that contains the modules in the module list and writes it to the specified file. Typically, the `GENERATE_LIBRARY` command is the last command you enter before ending the `CREATE_OBJECT_LIBRARY` session.

After you have generated the library, you end the `CREATE_OBJECT_LIBRARY` session by typing

**`QUIT`**

The following terminal dialog shows a simple `CREATE_OBJECT_LIBRARY` session in which a library containing the modules from a single object file is created.

```

/create_object_library <----- Begin the session.

COL/add_modules library=lgo <----- Add the modules on file
                                LGO to the library list.

COL/generate_library library=mylib <-- Generate the new library on
                                file MYLIB.

COL/quit <----- End the session.

```

## Creating Object Libraries

Creating a new object library involves the following sequence of steps:

1. Begin the `CREATE_OBJECT_LIBRARY` session by entering a `CREATE_OBJECT_LIBRARY` command.
2. Specify the modules to be included in the library by entering an `ADD_MODULES` command.

3. Generate the library by entering a `GENERATE_LIBRARY` command.
4. End the session by typing `QUIT`.

To specify modules to be included in a new or existing library while in a `CREATE_OBJECT_LIBRARY` session, you use the `ADD_MODULES` command. This command has the form

**`ADD_MODULES LIBRARY=file-list`**

where `file-list` specifies one or more files containing modules to be added to the library.

The files you specify can be object files produced by the loader or they can be SCL procedures or other libraries. `CREATE_OBJECT_LIBRARY` adds all the modules from the specified files to the object library.

If you want to add selected modules from a file to a library, you can specify the `MODULES` parameter as follows:

**`ADD_MODULES LIBRARY=file-list MODULES=mod-list`**

This command causes the `CREATE_OBJECT_LIBRARY` to add only the modules specified in `mod-list` from the specified files to the module list.

For example, the following command, entered during an object library session, adds three modules to an object library:

```
/COL add_modules library=bin_file module=(bin1, bin2, bin3)
```

The modules `BIN1`, `BIN2`, and `BIN3` on file `BIN_FILE` are added to the module list.

Remember that after you add the modules to `CREATE_OBJECT_LIBRARY`'s module list, you must enter a `GENERATE_LIBRARY` command to actually create the library.

## Modifying Object Libraries

`CREATE_OBJECT_LIBRARY` enables you to modify existing object libraries. These modifications include adding, replacing, and deleting modules in existing object libraries. The following list summarizes the commands you need to perform these modifications:

<code>ADD_MODULES</code>	Adds one or more modules to a library.
<code>REPLACE_MODULES</code>	Replaces one or more modules in a library.
<code>DELETE_MODULES</code>	Deletes one or modules from a library.

Remember that these commands alter the contents of the module list, rather than an actual object library. When you begin a `CREATE_OBJECT_LIBRARY` session to modify an existing library, the first step is to add all the modules in that library to the module list. You can do this with the command

```
ADD_MODULES LIBRARY=library
```

where the specified library is the one you want to modify. You can then use the `ADD_MODULES`, `REPLACE_MODULES`, and `DELETE_MODULES` commands to alter the library list. Then, after you make all the desired changes to the module list, you use the `GENERATE_LIBRARY` command to replace the existing library.

### Adding Modules

You use the `ADD_MODULES` command to add modules to an existing library. For example, the following commands add two modules from file `LGO` to a library named `ALIB`:

```
/COL add_modules library=alib <---- Add modules from library
                                ALIB to module list.

/COL add_modules library=lgo <----- Add modules from file LGO to
                                module list.

/COL generate_library file=alib <-- Rewrite existing library.

/COL quit <----- End session.
```

### Replacing Modules

To replace modules in the module list, you use the `REPLACE_MODULES` command. This command has the form

```
REPLACE_MODULES FILE=file-list MODULES=mod-list
```

where `file-list` specifies one or more files containing replacement modules, and `mod-list` specifies one or more modules to be replaced. You can omit the `MODULES` parameter, in which case all of the modules in the specified files are used as replacement modules.

When `CREATE_OBJECT_LIBRARY` executes a `REPLACE_MODULES` command, it begins by searching the module list for a module having the same name as the first module in the first file in the `file-list` (or in the `mod-list`, if you specified the `MODULES` parameter). If a match is found, the replacement module replaces the existing module.

If no match is found after searching the entire module list, the replacement module is disregarded (it is NOT added to the library), and processing continues with the next module in the file (or in the mod-list, if you specified the MODULES parameter).

The following REPLACE\_MODULES command, entered in a CREATE\_OBJECT\_LIBRARY session, specifies three modules to be replaced:

```
/COL replace_modules file=bin_file modules=(moda, modb, modc)
```

CREATE\_OBJECT\_LIBRARY first searches the module list for a module named MODA. If MODA is found, it is replaced by MODA from file BIN\_FILE. If MODA is not found, no replacement occurs. In either case, CREATE\_OBJECT\_LIBRARY then searches the module list for modules MODB and MODC, and replaces them if they are found.

## Deleting Modules

You delete modules from the module list by specifying a DELETE\_MODULES command in a CREATE\_OBJECT\_LIBRARY session. This command has the form

```
DELETE_MODULES MODULES=mod-list
```

where mod-list specifies one or more modules to be deleted from the module list. CREATE\_OBJECT\_LIBRARY searches the module list for the specified modules and deletes the ones it finds.

For example, the command

```
/COL delete_modules modules=(mod1, mod4, mod6)
```

deletes modules mod1, mod4, and mod6 from the library list.

After you have specified all the modules to be added, deleted, or replaced in the module list, you create a library that contains the modules currently in the module list by entering the command

```
GENERATE_LIBRARY LIBRARY=library
```

The library you specify can be either an existing library or a new one. If you specify an existing library, it is replaced.



## Displaying Information About Object Libraries

You can display a list of the modules in any library or object file by entering a `DISPLAY_OBJECT_LIBRARY` command. This command has the form:

```
DISPLAY_OBJECT_LIBRARY LIBRARY=file-name
```

where `file-name` specifies an object file or a library file. The `DISPLAY_OBJECT_LIBRARY` command lists all the modules in the specified library or object file, as well as the date and time each module was placed in the library. (You can request more information through additional parameters, as described in the Object Code Management manual.)

The `DISPLAY_OBJECT_LIBRARY` command is an SCL command, rather than a library generator command. That means that you can enter it either within or outside of a `CREATE_OBJECT_LIBRARY` session. Thus, you can use this command to list the modules in a library you are currently updating, in any of the object files you are using to update the library, or in any other object file or library.

For this example, assume the following:

Object file `BIN_FILE_1` contains modules A, B, and C

Object file `BIN_FILE_2` contains modules D, E, and F

The command

```
DISPLAY_OBJECT LIBRARY LIBRARY=BIN_FILE_1
```

displays a list similar to the following:

A	OBJECT MODULE	08:15:29	1983-06-04
B	OBJECT MODULE	08:15:27	1983-06-04
C	OBJECT MODULE	15:33:04	1983-06-03

The list shows the module name, module type, and the time and date the module was placed in the library.

Now suppose we execute the following `CREATE_OBJECT_LIBRARY` session:

```
/create_object_library
/col add_modules files=(bin_file_1,bin_file_2) modules=(a,b,e,f)
/col generate_library library=mylib
/col display_object_library library=mylib
/col quit
/display_object_library library=mylib
```

This session creates an object library that contains four modules. Two `DISPLAY_OBJECT_LIBRARY` commands are entered: one inside the `CREATE_OBJECT_LIBRARY` session and one after the session is ended. Both commands display the same list:

A	LOAD MODULE	08:53:04	1983-06-04
B	LOAD MODULE	08:53:30	1983-06-04
E	LOAD MODULE	08:54:10	1983-06-04
F	LOAD MODULE	08:54:26	1983-06-04

## Example of Creating and Modifying an Object Library

The following is a simple example of creating and modifying a library. We begin with the following main program subroutines:

```
PROGRAM MAIN
PRINT *, ' IN MAIN PROGRAM'
CALL SUBA
CALL SUBB
END
```

```
SUBROUTINE SUBA
PRINT *, ' IN SUBA'
RETURN
END
```

```
SUBROUTINE SUBB
PRINT*, ' IN SUBB'
RETURN
END
```

The main program is stored on file `MAIN_SOURCE` and both subroutines are stored on file `SUB_SOURCE`. We wish to create a library containing modules `SUBA` and `SUBB`.

First, we compile the main program and subroutines with the following commands:

```
/fortran i=main_source b=main_bin
/fortran i=sub_source b=sub_bin
```

File `MAIN_BIN` contains the object code for the main program, and file `SUB_BIN` contains the object code for subroutines `SUBA` and `SUBB`.

We now use `CREATE_OBJECT_LIBRARY` to create the library:

```

/create_object_library
COL/add_modules library=sub_bin modules=(suba,subb) <-- Add
                                modules SUBA and
                                SUBB from file
                                SUB_BIN.

COL/generate_library library=sub_lib <-- Generate the library on
                                file SUB_LIB.

COL/quit

```

We can display a list of modules in the library by entering the following command:

```
/display_object_library library=sub_lib
```

This command displays the following list:

SUBA	OBJECT MODULE	09:58:04	1983-10-18
SUBB	OBJECT MODULE	09:58:14	1983-10-18

The following command loads the program, searches library `SUB_LIB` to satisfy external references, and begins execution.

```
/execute_task file=main_bin library=sub_lib
```

This command does the following:

Loads the program on file `MAIN_BIN` into memory. (This file contains the object code for the main program.)

Satisfies the external references to `SUBA` and `SUBB`. Since `SUBA` and `SUBB` were not loaded from file `MAIN_BIN`, the loader searches for them in the library `SUB_LIB`.

Begins execution of the loaded program. The program prints the following:

```

IN MAIN PROGRAM
IN SUBA
IN SUBB

```

## EXAMPLE OF OBJECT LIBRARY

We now wish to update library SUB\_LIB. We will replace module SUBA with a new version and add a new module named NEWSUB. The new version of SUBA contains a call to NEWSUB:

```
SUBROUTINE SUBA
PRINT *, ' IN NEW SUBA'
CALL NEWSUB
RETURN
END
```

```
SUBROUTINE NEWSUB
PRINT *, ' IN NEWSUB'
RETURN
END
```

Assume that the object code for SUBA is on file NEW\_BIN and the object code for NEWSUB is on file NEW\_SUBS.

The CREATE\_OBJECT\_LIBRARY session is as follows:

```
/create_object_library
COL/add_modules library=sub_lib <-- Add the modules from the
                                existing library to the module
                                list.

COL/replace_modules library=new_bin module=suba <-- Replace
                                module SUBA on the library
                                with the module of the same
                                name from file NEW_BIN.

COL/add_modules library=new_subs module=newsub <-- Add module
                                NEWSUB from file
                                NEW_SUBS.

COL/generate_library library=new_lib <-- Generate a new library
                                named NEW_LIB.

COL/quit
```

We can list the modules in the new library with the command

```
/display_object_library library=new_lib
```

SUBA	OBJECT MODULE	10:15:22	1983-10-18
SUBB	LOAD MODULE	10:15:30	1983-10-18
NEWSUB	OBJECT MODULE	10:16:06	1983-10-18

The following command executes the program using modules from the new library:

```
/execute_task file=main_bin library=new_lib
```

Output from the program is as follows:

```
IN MAIN PROGRAM
IN NEW SUBA
IN NEWSUB
IN SUBB
```

## Summary of Using Object Libraries

Before a compiled program can be executed, it must be processed by a system utility called the loader. The loader uses object libraries during the loading process. Before you learn about object libraries, you should understand the loading process.

The simplest type of load is performed by the name call command. The familiar command LGO is a form of the name call command. A name call command consists of the name of the file containing the compiled object code. When you specify a name call command, the loader performs the following sequence of steps:

1. Loads the compiled program units from the specified file into memory.
2. Supplies the loaded program with the addresses of all routines called by the program. This process is known as satisfying external references.
3. Begins execution of the loaded program.

The name call command loads program units from a single file. If a program calls routines that reside on different files, you can use the EXECUTE\_TASK command to load the required routines. Like the name call command, the EXECUTE\_TASK command loads and executes a program, but it provides additional options. The FILES parameter on this command specifies files from which routines are to be loaded.

After the loader has completed a load operation, it produces a load map that consists of a summary of the load operation. The load map contains such information as the length and location in memory of each loaded program unit. When used with a memory dump and object listing, the load map can be a useful debugging tool for the experienced programmer.

In the process of locating entry points to satisfy external references, the loader searches a special type of file known as an object library.

An object library is a specially-formatted file containing object programs. When the loader satisfies external references, it quickly searches the available libraries for any referenced modules and loads them into memory with the referencing module.

Object libraries provide the following advantages:

- The loader automatically searches the libraries that are available to a program, and loads the required modules.
- With libraries, the loader loads ONLY those modules that are required by the program, regardless of how many modules are stored in the library.
- Modules on a library can be shared by other programs.

When the loader satisfies external references, it first searches for the referenced entry points in modules already loaded. If unsatisfied external references remain after this search, the loader searches the available object libraries for the referenced entry points. Upon finding a referenced entry point in a library, the loader loads the routine containing that entry point and provides the calling modules with the addresses of the entry points.

You can make a library available for loader searching in two ways. The first way is to declare the library in the LIBRARY parameter of an EXECUTE\_TASK command. This adds the library to the local library list. The local library list is available only for the current EXECUTE\_TASK command.

The second way of making a library available for loader searching is to declare it in the ADD\_LIBRARY parameter of a SET\_PROGRAM\_ATTRIBUTE command. This adds the library to the program library list. The program library list is available to the loader until you either delete the libraries or end the terminal session.

In addition to the libraries you have declared in an EXECUTE\_TASK or SET\_PROGRAM\_ATTRIBUTE command, the FORTRAN compiler and NOS/VE system provide other libraries that contain routines required by most programs.

When the loader satisfies external references, it searches the available libraries in the following predefined order:

1. Local library list (as specified by the EXECUTE\_TASK command).
2. Object libraries specified by the compiler.
3. Job library list (as specified by the SET\_PROGRAM\_ATTRIBUTE command).
4. Job debug library list (searched only if you have turned on debug mode).
5. NOS/VE task services library.

Within each of the above lists, the loader searches the libraries in the order of their occurrence in the list.

You use the CREATE\_OBJECT\_LIBRARY utility to create and modify object libraries. To create a new library, or to modify an existing one, conduct a CREATE\_OBJECT\_LIBRARY session as follows:

1. Begin the session by entering the command  
CREATE\_OBJECT\_LIBRARY.
2. Enter commands to add, replace, or delete modules in the library.  
Commands are:
  - ADD\_MODULES to add modules to a new or existing library
  - REPLACE\_MODULES to replace modules in an existing library
  - DELETE\_MODULES to remove modules from an existing library
3. Generate the new library, or replace the existing one, by entering the command GENERATE\_LIBRARY.
4. End the CREATE\_OBJECT\_LIBRARY session by typing the command QUIT.





# Index

---

## A

- ABORT\_FILE Debug feature 2-44
- Access violation error 2-7
- Address space 5-3
- ADD\_MODULE
  - subcommand 6-16, 17
- ANSI input/output methods 3-30
- Array initialization 4-14; 5-15
- Assignment statement,
  - replacement with DATA statement 4-14
- Attaching files 1-7
- ATTACH\_FILE SCL
  - command 1-7
- Automatic execution of
  - Debug commands 2-44

## B

- Batch job submission 1-29
- BINARY\_OBJECT FORTRAN
  - command parameter 1-26; 2-2
- Block IF structures 2-10
- Branching, efficient
  - techniques 4-14
- BUFFER IN statement 3-16
- BUFFER OUT statement 3-16

## C

- Catalog
  - Creating 1-11
  - Definition of 1-2
  - Displaying contents of 1-12
  - Working 1-14
- \$LOCAL 1-3
- \$USER 1-3
- CHANGE\_FILE\_ATTRIBUTES SCL command 1-20

- CHANGE\_PROGRAM\_VALUE
  - Debug command 2-28
- COLLECT\_TEXT SCL
  - command 1-30
- Common blocks
  - Effect on optimization 4-12
  - In virtual memory 5-16
- Common programming errors 2-7
- Common subexpressions 4-5
- Compiler control directives 2-26
- Compiling 1-25
- Conditional compilation
  - directives 2-26
- CONNEC subroutine call 1-23
- Constant subexpressions 4-4
- COPY\_FILE SCL command 1-3
- CREATE\_CATALOG SCL
  - command 1-11
- CREATE\_FILE\_PERMIT SCL
  - command 1-18
- CREATE\_FILE SCL
  - command 1-6
- CREATE\_OBJECT\_LIBRARY
  - utility
    - ADD\_MODULE
      - subcommand 6-16, 17
    - DELETE\_MODULE
      - subcommand 6-18
    - General description 6-14
    - GENERATE\_LIBRARY
      - subcommand 6-15, 18
    - REPLACE\_MODULE
      - subcommand 6-17
- C\$ Directives 2-26
- C\$ PARAM directive 1-35

## D

- DATA statement, use in
  - optimization 4-14
- Dead instructions 4-4

## Debug

- Command summary 2-51
- Example 2-46
- Facility 2-28
- Mode 2-32
- Session 2-31

## Debug commands

- Automatic execution of 2-44
  - CHANGE\_PROGRAM\_VALUE 2-40
  - DISPLAY\_BREAK 2-41
  - DISPLAY\_CALL 2-42
  - DISPLAY\_DEBUG\_ENVIRONMENT 2-43
  - DISPLAY\_VALUE 2-38
  - QUIT 2-40
  - RUN 2-37
  - SET\_BREAK 2-35
  - SET\_STEP\_MODE 2-36
  - Summary 2-51
- DEBUG FORTRAN command
- parameter 2-31
- Debugging 2-1
- Debugging aids
- Debug facility 2-28
  - Reference map 2-20
- DELETE\_BREAK Debug command 2-41
- DELETE\_FILE SCL command 1-10
- DELETE\_MODULE subcommand 6-18
- DETACH\_FILE SCL command 1-10
- Direct access files 3-18
- DISPLAY\_BREAK Debug command 2-41
- DISPLAY\_CALL Debug command 2-42
- DISPLAY\_CATALOG SCL command 1-12
- DISPLAY\_DEBUG\_ENVIRONMENT Debug command 2-43
- DISPLAY\_FILE\_ATTRIBUTES SCL command 1-20

## DISPLAY\_JOB\_STATUS

- SCL command 1-31
- DISPLAY\_LOG SCL command 2-3
- DISPLAY\_OBJECT\_LIBRARY SCL command 6-19
- DISPLAY\_PROGRAM\_ATTRIBUTES SCL command 2-3
- DISPLAY\_PROGRAM\_VALUE Debug command 2-38
- Divide fault error 2-7
- DO loops, optimization of 4-7, 11, 12

**E**

- EQUIVALENCE statement; effect on optimization 4-12
- ERROR FORTRAN command parameter 2-2
- EXECUTE\_TASK SCL command
- General description 1-27
  - Used to generate a load map 6-5
  - Used to load modules from files 6-4
  - Used to load modules from libraries 6-8, 9
  - Used to set debug mode 2-33
- Executing a FORTRAN program 1-26
- Execution time parameters 1-34
- EXPRESSION\_EVALUATION FORTRAN parameter 4-16
- CANONICAL option 4-16
  - General description 4-16
- MAINTAIN\_EXPRESSION option 4-16
- MAINTAIN\_PRECISION option 4-18
- Expressions, factoring 4-15

**F**

- Factoring expressions 4-15
- File access permission 1-18
- File interface subprograms 3-22
- File reference 1-4
- Files
  - Access permission 1-18
  - Attaching 1-7
  - Attributes 1-19; 3-3, 24
  - Copying 1-3
  - Creating 1-5
  - Cycles 1-17
  - Definition of 1-1
  - Deleting 1-9
  - FORTRAN direct access 3-18
  - Local 1-2
  - Names 1-4
  - NOS-NOS/VE file transfer 1-24
  - Permanent 1-2
  - Positioning 1-16
  - Random 3-16
  - Referencing 1-4
  - Sequential access 3-16
  - \$INPUT 1-21
  - \$OUTPUT 1-21
- Format specification 3-9
- FORMAT statement 3-9
- Formatted input/output 3-5
- FORTRAN command 1-20
- FORTRAN command parameters
  - BINARY\_OBJECT 1-26
  - DEBUG 2-31
  - EXPRESSION\_EVALUATION 4-16
  - INPUT 1-26
  - LIST 1-26; 2-21
  - LIST\_OPTIONS 2-21
  - ONE\_TRIP\_DO 4-18
  - OPTIMIZATION 4-18
- Free space in virtual memory 5-1

**G**

- GETCVAL subroutine call 1-36
- GET\_FILE SCL command 1-24

**H**

- HELP SCL command 2-24

**I**

- Identity instructions 4-3
- Indefinite value 2-4
- Infinite value 2-5
- INPUT FORTRAN command
  - parameter 1-26
- Input/output methods
  - Buffer 3-16
  - Comparison of 3-11
  - Examples of 3-11
  - File interface 3-22
  - Formatted 3-9
  - List directed 3-5
  - Mass storage 3-24
  - Namelist 3-7
  - Selecting a method 3-26
  - Unformatted 3-15
- Input/output units 3-2
- Integer arithmetic 2-6
- Interactive
  - Debugging 2-28
  - Input/output 1-20

**L**

- LGO command 1-27; 6-3
- Library search order 6-12
- Libraries (see Object Libraries)
- List directed input/output 3-5
- LIST FORTRAN command
  - parameter 1-26; 2-21
- LIST\_OPTIONS FORTRAN
  - command parameter 2-21
- Listing control directives 2-27
- Load map 1-29; 6-5
- Loader 6-1, 8

**Loading**  
   From files 1-28; 6-4  
   From libraries 1-28; 6-4  
   Name call 6-3  
**Local files** 1-3  
**Locality of reference**  
   General description 5-9  
   Suggestions for improving 5-14  
**LOGIN command** 1-30  
**Loops**  
   Combining 4-12  
   Removing operations from 4-7  
   Unrolling 4-11

## M

**Mass storage input/output** 3-24  
**Memory preset value** 1-29; 2-3  
**Message mode** 2-26  
**MESSAGES online manual** 2-26  
**Mixed mode arithmetic** 4-13  
**Modular programming**  
   structure 2-9

## N

**Name call loading** 6-3  
**Namelist input/output** 3-7  
**Nonstandard usages** 2-16

## O

**Object libraries**  
   Creating 6-15  
   Displaying information  
     about 6-19  
   Example of 6-20  
   General description 6-7  
   Modifying 6-16  
   Search order 6-12  
   Summary 6-23  
**ONE\_TRIP\_DO FORTRAN**  
   command parameter 4-18  
**OPEN statement** 3-2, 18

**Opening files** 3-2  
**OPTIMIZATION FORTRAN**  
   command parameter 2-32  
**Optimizing**  
   General description 4-1  
   Summary 4-21

## P

**Page**  
   In virtual memory 5-3  
   Replacement algorithm 5-4  
   Swap 5-4  
   Thrashing 5-8  
**Parameters**  
   FORTRAN 1-26  
   On execution command 1-34  
**Permanent file cycles** 1-17  
**Permanent files** 1-3  
**Presetting memory** 1-29; 2-3  
**PRINT statement** 3-5

## Q

**QUIT Debug command** 2-40

## R

**Random access** 3-16  
**READ statement**  
   Formatted 3-9  
   List directed 3-5  
   Namelist 3-8  
   Unformatted 3-15  
**Record key** 3-17  
**Reference map**  
   Generating 2-21  
   Statement labels map  
     section 2-22  
   Variables map section 2-21  
**REPLACE\_FILE SCL**  
   command 1-25

**REPLACE\_MODULE**  
 subcommand 6-17  
**REQUEST\_TERMINAL SCL**  
 command 1-23  
**REWIND\_FILE SCL**  
 command 1-16  
**RUN Debug command** 2-37  
 Runtime error conditions 2-4

## S

SCLCMD subroutine call 1-33  
 Sequential access files 3-16  
**SET\_BREAK Debug**  
 command 2-35  
**SET\_FILE\_ATTRIBUTES SCL**  
 command 1-20  
**SET\_MESSAGE\_MODE SCL**  
 command 2-2, 26  
**SET\_PROGRAM\_ATTRIBUTES**  
 SCL command  
   General description 2-32  
   Used to set debug mode 2-32  
   Used to declare libraries  
     6-10, 12  
   Used to specify  
     **ABORT\_FILE** 2-44  
**SET\_STEP\_MODE Debug**  
 command 2-36  
**SET\_WORKING\_CATALOG SCL**  
 command 1-15  
 Short forms of SCL  
 commands 1-38  
 Statement labels map 2-22  
 Subcatalogs 1-10  
**SUBMIT\_JOB SCL**  
 command 1-31

Subprogram traceback list 2-42  
 System Command Language  
 (SCL) 1-1

## T

Traceback list 2-42

## U

Unformatted input/output 3-15

## V

Variables map 2-21  
 Virtual memory  
   General description 5-1  
   Programming guidelines 5-8

## W

Working catalog 1-15  
**WRITE statement**  
   Formatted 3-9  
   List directed 3-5  
   Namelist 3-8  
   Unformatted 3-15  
 \$ASIS file position 1-16  
 \$BOI file position 1-16  
 \$EOI file position 1-16  
 \$INPUT file 1-21  
 \$LOCAL catalog 1-3  
 \$OUTPUT file 1-21  
 \$USER catalog 1-3



**FORTRAN for NOS/VE, Topics for FORTRAN Programmers Usage 60485916 A**

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

**Who Are You?**

- Manager
- Systems Analyst or Programmer
- Applications Programmer
- Operator
- Other \_\_\_\_\_

**How Do You Use This Manual?**

- As an Overview
- To Learn the Product System
- For Comprehensive Reference
- For Quick Look-up

**Do You Also Have?**

- FORTRAN Tutorial
- FORTRAN Language Definition Usage

What other programming languages do you use? \_\_\_\_\_

**How Do You Like This Manual?** Check those that apply.

- | Yes                      | Somewhat                 | No                       |   |
|--------------------------|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Is the manual easy to read (print size, page layout, and so on)?                                      |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Is it easy to understand?   |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Is the order of topics logical?   |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Are there enough examples?  |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Are the examples helpful? ( <input type="checkbox"/> Too simple <input type="checkbox"/> Too complex) |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Is the technical information accurate?  |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Can you easily find what you want?  |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Do the illustrations help you?  |
| <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | Does the manual tell you what you need to know about the topic?                                       |

**Comments?** If applicable, note page number and paragraph.

Would you like a reply?  Yes  No

Continue on other side

**From:**

Name \_\_\_\_\_ Company \_\_\_\_\_

Address \_\_\_\_\_ Date \_\_\_\_\_

\_\_\_\_\_ Phone No. \_\_\_\_\_

Please send program listing and output if applicable to your comment.



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO 8241      MINNEAPOLIS MINN

POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**

Publications and Graphics Division

P.O. BOX 3492

Sunnyvale, California 94088-3492



Comments (continued from other side)

F010