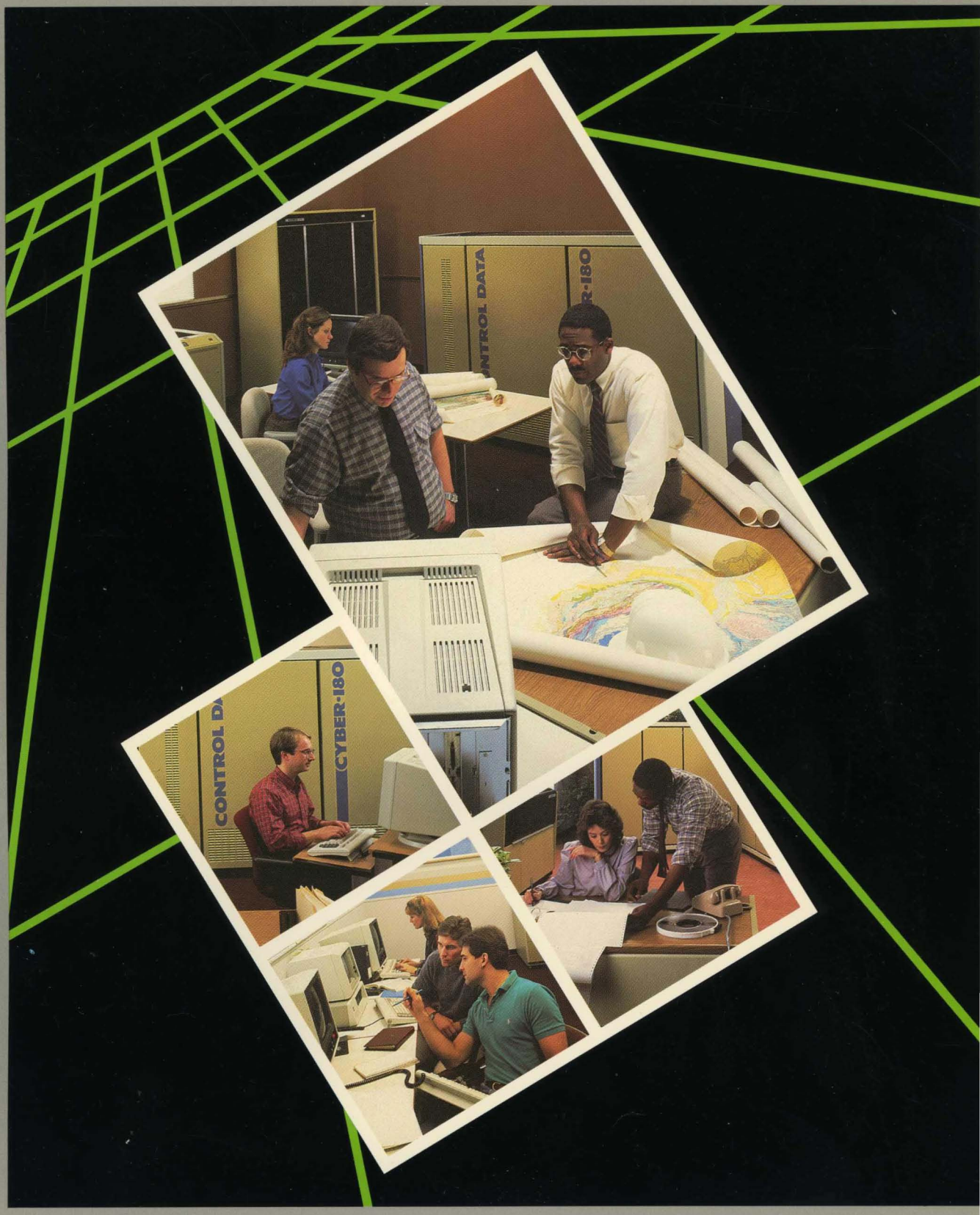
# Migration
# From NOS to NOS/VE

CONTROL DATA




## Tutorial/Usage

60489503

# Migration
# From NOS to NOS/VE

## Tutorial/Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

Publication Number 60489503

# Manual History

| Revision | System Version/ PSR Level | Date |
|---|---|---|
| A | 1.0.2/589 | October 1983 |
| B | 1.1.1/613 | June 1984 |
| C | 1.1.2/630 | March 1985 |
| D | 1.1.3/644 | September 1985 |
| E | 1.1.4/649 | January 1986 |
| F | 1.2.2/678 | June 1987 |

**This revision:**

Revision F was published in June 1987. This revision reflects the NOS/VE operating system at release 1.2.2, PSR level 678. This revision documents the following new software capabilities:

Migration by using tape migration commands

The permanent file transfer facility

Miscellaneous technical and editorial changes and corrections have been made throughout the manual.

# Contents

## Migrating Programs



## Appendixes

# About This Manual

# About This Manual

This manual is intended to help you migrate your files and programs from a CYBER computer operating under the CONTROL DATA® Network Operating System (NOS) to a CYBER 180 computer operating under the CDC® Network Operating System/Virtual Environment (NOS/VE) executing in the dual state configuration. Dual state NOS/VE has a NOS front end. Migrating refers to the process of transferring files from NOS to NOS/VE and changing the files or program source code so that they will work on NOS/VE.

The manual provides an overview of migration requirements and resources available for migration. The kinds of information provided are:

Detailed information about the differences between COBOL and FORTRAN on NOS and NOS/VE

Detailed information about file structure differences on the two systems

An overview of NOS/VE facilities available for migrating files and examples of using some of the facilities

Simple explanations of NOS/VE features and command language essential in starting to migrate applications

## Audience

This manual assumes that you are a programmer who is familiar with NOS. Also, you have COBOL or FORTRAN applications running on NOS that you wish to migrate to NOS/VE. For any feature discussed about COBOL, FORTRAN, or files in general, you are assumed to be familiar with the feature on NOS.

This manual assumes that you have NOS/VE manuals available. The related manuals diagram on the back of the title page shows NOS/VE manuals that you might need. For a complete list of NOS/VE publications and information on how to order them, see the NOS/VE System Information Manual (the default online manual for the EXPLAIN command).

## Organization

This manual focuses on information that is important in migrating files and programs. The information in the manual logically divides into three major parts: using the NOS/VE System Command Language (SCL) and NOS/VE commands from a NOS user's viewpoint, migrating files, and migrating programs. The information in the chapters is interdependent. You might actually want to look over the information on migrating programs first; however, some explanations in those chapters depend on your familiarity with background information about NOS/VE files provided in preceding chapters. The manual organizes information as follows:

Using SCL Commands from a NOS User's Viewpoint

These chapters provide information about using NOS/VE commands. Chapter 1 describes logging in and out. Chapter 2 discusses SCL conventions for commands, for SCL names, and for specifying parameters in SCL commands. Chapter 3 gives an overview of the permanent file mechanism to help you establish and locate files on NOS/VE. Chapter 4 indicates similar NOS and NOS/VE commands and discusses commonly used NOS/VE commands. Chapters 5 through 9 present more in-depth information about SCL commands and concepts. Topics covered in these chapters include SCL job structure; SCL procedures; compiling, loading, and executing programs under NOS/VE; NOS/VE object libraries; and batch job submission.

Migrating Files

These chapters focus on handling files. Chapter 10 indicates the differences in NOS and NOS/VE file organizations and record types, and provides an introduction to the NOS/VE file interface. Chapter 11 discusses general NOS/VE facilities that migrate files. Chapter 12 describes the FORTRAN and COBOL File Migration Aid.

Migrating Programs

These chapters deal directly with coding changes required to migrate a program. Chapter 13 discusses ways to approach FORTRAN or COBOL program migration. Chapter 14 discusses migrating FORTRAN programs. Chapter 15 discusses migrating COBOL programs. Chapter 16 discusses migrating APL workspaces. Chapter 17 discusses migrating Pascal programs.

Additional information is available in the following appendixes:

A. A glossary of terms relating to NOS/VE usage.

B. Character Sets and Collating Sequences

Lists all characters in the 7-bit ASCII character set and lists predefined collating sequences available on NOS/VE.

C. NOS and NOS/VE Similarities/Differences Summary

Lists computer system differences for the computer systems supporting only NOS and the dual-state systems supporting both NOS and NOS/VE.

D. Unsupported ANSI COBOL Features

Lists features described in ANSI 74 COBOL Standard but not implemented in NOS/VE COBOL.

E. FORTRAN Default FIT Field Values

## Submitting Comments

The last page of this manual is a comment sheet. Please use this comment sheet 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
Technology and Publications Division
P. O. Box 3492
Sunnyvale, CA 94088-3492

Please include this information with your comments:

The manual title, publication number, and revision level (for this manual, Migration From NOS to NOS/VE Tutorial/Usage, 60489503 F)

Your system's PSR level (if you know it)

Your name, your company's name and address, your work phone number, and whether you want a reply.

Also, if you have access to SOLVER, the CDC online facility for reporting problems, you can use it to submit comments about this manual. When SOLVER prompts you for a product identifier for your report, specify FA8.

## In Case of Trouble

Control Data's Central Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

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

or you can call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

## Conventions

UPPERCASE — In command and statement formats, uppercase indicates words or keywords that must appear exactly as shown. Uppercase is often used in examples of NOS control statements.

lowercase — In command and statement formats, lowercase indicates generic terms that represent the words or symbols that you supply. Lowercase is often used in examples of NOS/VE commands.

# Using SCL and NOS/VE Commands From a NOS User's Viewpoint

# Logging In and Out 1

# Logging In and Out 1

Your dual state CYBER computer can process with either NOS or NOS/VE. You can directly log in to either operating system by entering your usual family, user name, and so forth. Then specify the application you want:

For NOS, specify: IAF

For NOS/VE, specify: VEIAF

You can change processing from one operating system to another. When you change, the system terminates processing with one system and initializes processing with the other.

To switch from NOS to NOS/VE, specify: HELLO,VEIAF

The system associates your current NOS family and user name with your NOS/VE job.

To terminate processing on NOS/VE or to switch to processing on NOS, first you always specify: LOGOUT

The system then prompts for application. The choices are:

To begin processing on NOS, specify: IAF

The system associates your current NOS family and user name with your NOS/VE job.

To terminate processing, specify: BYE

To log in to NOS with a different user name, specify: HELLO

To begin a new session on NOS/VE, specify: VEIAF

All commands can be specified in either uppercase or lowercase letters. To the system, the commands:

logout and LOGOUT

are equivalent. This manual shows commands in uppercase letters to make them stand out in text.

At first, you may want to use your familiar text editor and prepare your NOS/VE programs, procedures, and batch jobs on NOS and transfer them to NOS/VE for processing. If your job has errors, you can return to NOS to make corrections and transfer the job file back to NOS/VE to run again. Special commands for transferring text files from one system to the other are available on NOS/VE. These commands are described in the common NOS/VE commands discussion and in the transferring files discussion.

There is one caution about switching from NOS to NOS/VE. Before logging in to NOS/VE, make permanent all NOS files that you want to keep (use the SAVE, REPLACE, or DEFINE commands). All your local files on NOS disappear when you login to NOS/VE. You can transfer only permanent files (either direct or indirect access files) to NOS/VE.

Optionally, you can prepare or edit program, procedure, and job files on NOS/VE by using the Full Screen Editor for NOS/VE. For information about the editor, see the File Editor Tutorial/Usage manual.

# Conventions for Commands, Names, and Parameters 2

# Conventions for Commands, Names, and Parameters 2

The System Command Language (SCL) controls job processing on NOS/VE. All NOS/VE commands follow SCL conventions. This guide introduces SCL conventions as follows:

Abbreviation convention for SCL commands

Specifying parameters in SCL commands

' Apostrophe for alphabetic or alphanumeric literals (called strings)

" Quotation marks for comments

.. Ellipsis for continuation

Rules for SCL names (file names, variable names, and so forth)

## Abbreviation Convention for SCL Commands

SCL commands have a verb-object structure. A command begins with a verb that indicates the action. The object or object and modifiers follow the verb. Commonly used verbs are:

CHANGE
CREATE
DISPLAY
SET

Examples of commands are:

COPY_FILE
SET_FILE_ATTRIBUTE or SET_FILE_ATTRIBUTES (Both singular and plural forms are valid.)

The underline character in the command separates the words in the command. Commands can be up to 31 characters in length.

The convention (with a few exceptions) for abbreviating commands is to use the first three characters of the verb and the first character of each additional word. Examples of commands and valid abbreviations are:

| | |
|---|---|
| COPY_FILE | COPF |
| CREATE_FILE | CREF |
| CREATE_INTERSTATE_CONNECTION | CREIC |
| SET_FILE_ATTRIBUTE | SETFA |

## Specifying Parameters in SCL Commands

Like NOS commands, NOS/VE commands consist of the command name and parameters. The following equivalent commands show differences between NOS and NOS/VE commands:

| NOS | NOS/VE |
|---|---|
| ATTACH,FILEA/M=W. | attach_file, $user.filea, access_mode=(read write execute)<br>attach_file $user.filea access_mode=(read write execute)<br>attach_file file=$user.filea access_mode=(read,write,execute) |

The commands attach your own permanent file FILEA with read, write, and execute permission. The differences are:

Either a space or a comma can be a parameter separator.

Either position or optional name can identify a parameter.

A parameter can be a list of values.

A file name becomes a file reference on NOS/VE.

A period terminator is not used on SCL commands.

## Space or Comma as Parameter Separator

Either a space or a comma can separate parameters. One or more spaces are equivalent to a space. Optionally, a comma can be preceded or followed by a space or spaces. The following commands are equivalent:

```
/attach_file,file=$user.filea,access_mode=(read,write)          Uses commas.

/attach_file, file=$user.filea, access_mode=(read, write)       Uses commas and spaces.

/attach_file   file=$user.filea   access_mode=(read   write)    Uses spaces.
```

Because spaces are separators, they are not ignored as they are in NOS commands. For example, consider equivalent commands that request the sum of 12 + 34:

```
NOS command       /DISPLAY, 12 + 34.
Results:               46    56B

NOS/VE command    /display_value 12 + 34
Results:          -- ERROR -- Expecting value, found '+' for parameter OUTPUT.

NOS/VE command    /display_value 12+34
Results:           46
```

On NOS, the spaces are ignored; on NOS/VE, the spaces separate parameters and result in an error in the first NOS/VE example.

## Identifying Parameters by Name or Position

You can specify parameters with or without the parameter name. Equivalent examples of the GET_FILE (GETF) command show both ways. (The commands obtain a copy of a NOS permanent file NOSFILE and transfer it to NOS/VE as file NVEFILE.)

```
/get_file  to=nvefile  from=nosfile    Using names TO and FROM
/get_file  nvefile  nosfile            Using position
```

When you specify parameters by position, the parameters are position (order) dependent, as indicated in the format of the command. When you specify the name, position does not matter. A parameter name, however, can mark a position so that you can list subsequent parameters in the positional format. More examples of the GETF command explain these situations.

The first five parameters of GET_FILE are:

```
TO  or  T
FROM  or  F
DATA_CONVERSION  or  DC
USER  or  U
PASSWORD  or  PW
```

You can use either the full parameter name or the abbreviation in the command.

The GET_FILE (GETF) command format with the first five parameter positions is as follows:

```
Parameter
Position---> 1          2         3         4       5
Format:  GETF T=nosvefile F=nosfile DC=option U=owner PW=pswd
```

The following three forms of the GETF command are equivalent: They transfer file FILEA of user OWNERX to your working area on NOS/VE:

```
Example A.    /get_file to=filea user=ownerx password=okay
Example B.    /get_file filea,,,ownerx,okay
Example C.    /get_file filea user=ownerx okay
```

Example A illustrates the parameter name specification.

Example B illustrates the positional specification. FILEA designates the value of the first parameter. The first comma following FILEA is a parameter separator. The next two commas indicate omitted parameters (parameters 2 and 3); therefore, OWNERX is the value of parameter 4 (USER). OKAY is the value of parameter 5.

Example C illustrates using mixed parameter name and positional form. FILEA is the value of parameter 1. The U parameter name positions at parameter 4 so that parameter 5 can follow with no parameter name specified.

## Abbreviation Convention for Parameters

Parameters have a complete name and an abbreviated form. In the complete name, the underline character (_) separates words in the name. Usually, the abbreviation is the first letters of each word in the parameter name. Examples:

| Full Parameter Name | Abbreviation |
|---|---|
| ACCESS_MODE | AM |
| FILE | F |
| TERMINATION_ERROR_LEVEL | TEL |

When specifying a parameter, you can specify either the complete or abbreviated form.

The following pairs of command specifications are equivalent:

```
/attach_file file=$user.transaction_file  access_mode=(read write)
/attach_file f=$user.transaction_file  am=(read write)
```

Attaches your permanent file TRANSACTION_FILE with access to read and write to the file.

```
/fortran input=$user.sourcfil binary_object=binfile
/fortran i=$user.sourcfil b=binfile
```

Calls the FORTRAN compiler and specifies the source code to be read from file your permanent file SOURCFIL and the object code to be written to local file BINFILE.

## Parameters as Value Lists

A parameter can specify a single value or a list of values (called a value list). The value list can be a series of values separated by a comma or a space and enclosed in parentheses. For example, the DETACH_FILE command has the following form:

```
DETACH_FILE  FILE=file  or  DETACH_FILE  FILE=(list of files)
```

The following example shows the value list consisting of files: FILE1, FILE2,FILE3 and FILE4:

```
/detach_file  file=(file1,file2,file3,file4)    Name identifies the parameter.

/detach_file (file1,file2,file3,file4)          Position identifies the parameter.
```

The ACCESS_MODE (AM) parameter of the ATTACH_FILE command can specify a list of values to gain desired access. The possible modes are:

```
READ        SHORTEN
APPEND      WRITE
MODIFY      ALL
EXECUTE
```

To attach your permanent file FILEA with read and write permission, specify the AM parameter as follows:

```
/attach_file  file=$user.filea  access_mode=(read  write)
```

The list must be enclosed in parentheses.

## Specifying Parameters as a Range

Some parameters can be specified as a range of values. The form for specifying range is:

```
n..m
```

The n indicates the lowest value, the ellipsis (..) indicates the range, and m indicates the highest value.

For example, the command to pass a range of numbers to a FORTRAN program could appear as follows (the object code is on file LGO):

```
LGO 1000..1999
```

This example assumes that the program is coded with parameter interface subroutines to obtain and use the numbers 1000 through 1999 in the run. (See the discussion on Using the NOS/VE Parameter Interface Subprograms in chapter 14 for more information.)

### Parameter Value Types

The System Command Language (SCL) ensures that each parameter is set to its proper type. The common types of values for parameters are:

| Type | Example |
|---|---|
| Name or key | ALL<br>N2000 |
| Integer | 100<br>-5000 |
| String | 'This is a string'<br>'Names don''t have apostrophes, strings do' |
| Boolean | YES NO<br>TRUE FALSE<br>ON OFF |
| File | MY_FILE<br>$USER.MY_FILE<br>$LOCAL.MY_FILE<br>(File references are described in chapter 3.) |

## ' Apostrophe for Alphabetic or Alphanumeric Literals

Apostrophes enclose alphabetic or alphanumeric literals in SCL commands. In SCL terminology, these literals are called strings. Examples are:

```
/put_line ' This is a string.'

/put_line ' Use two apostrophes '' within a string.'
```

Note that within a string, case is significant. That is, uppercase letters are not equivalent to lowercase letters (an A is not an a).

## " Quotation Marks for Comments

Quotation marks enclose comments in SCL command lines. An end-of-line also terminates a comment.

Comments can appear anywhere within a command where a space would be allowed. For example:

```
fortran  input=infile  "Diagnosing non-ANSI usage"  list=listfil
```

Comments can also appear on a separate line. For example:

```
"This is a comment line
```

## ..Ellipsis for Continuation

An ellipsis (two or more contiguous periods ..) at the end of a command indicates continuation. A command can be any length and can include several physical lines on your terminal. An ellipsis at the end of a physical line specifies continuation to the next line. This applies to comment lines as well. For example, the following two lines:

```
"This is ..
a comment line.
```

are the same as:

```
"This is a comment line.
```

For example, the SETFA (SET_FILE_ATTRIBUTE) command for file LOGFILE specifying a separate parameter on each line appears as follows:

```
/set_file_attributes  file=logfile ..
.. /file_content=list "Character data with carriage control" ..
.. /file_organization=sequential ..
.. /status=logstat
```

NOTE

Ellipsis also indicates a range of values (discussed earlier in this chapter).

## Rules for SCL Names

The names used for files, variables, and procedures in SCL commands:

- Must not exceed 31 characters in length
- Can be made from the following characters:

| | |
|---|---|
| Digits: | 0 through 9 |
| Letters: | a through z or A through Z |
| Other characters: | _ # $ @ [ \ ] ^ ` { \| } ~ |

- Must begin with a letter or one of the following: _ # $ @

The system does not distinguish between corresponding uppercase and lowercase letters in file, procedure, and variable names. That is, an A is also an a.

NOTE

Names of special system procedures, variables, files, or functions (including user-written functions) begin with the character $. System names usually have $ for the fourth character. User-defined variable names must not begin with $.

# Overview of the Permanent File Mechanism 3

# Overview of the Permanent File Mechanism 3

The NOS/VE permanent file mechanism differs from that of NOS. One difference is that the NOS concept of direct and indirect access files does not apply to NOS/VE. Also, NOS/VE organizes files hierarchically into catalogs. A catalog is a collection of files and other catalogs.

To reference a NOS/VE file, you use a file reference. A file reference can include a family name, user name, optional catalog name or names, file name, cycle, and file position. These are strung into a file reference like this:

```
:family.user.catalog.file.cycle.position
```

This manual assumes the use of a default family name, so the family name is not discussed. If you are using a family in NOS/VE that is different from the family you are using in NOS, you might need to use the SET_LINK_ATTRIBUTE command. The SCL System Interface Usage manual discusses this command. Also, some SCL commands, such as COPY_FILE, can reference files having a family name that is different from the one you logged in with. Make sure that the references to these files include the family name.

The topics discussed are:

Catalogs (every file is in a catalog)

File Position (usually at $BOI; you occasionally need to designate $EOI or $ASIS)

Permanent File Cycles (versions of a file)

Summary of Simple File References

## CATALOGS

Each NOS/VE file belongs to a catalog. Two catalogs are important to remember: local and master. Your local catalog contains temporary files and is referenced by using $LOCAL in place of a catalog name in a file reference.

Your master catalog contains all of your permanent files and subcatalogs. The name of your master catalog is your user name. You can also reference your master catalog by using $USER in place of a catalog name in a file reference.

When processing on NOS/VE, you must be aware of your working catalog. The working catalog is the catalog used if you don't specify a catalog on a file reference. The default working catalog is $LOCAL when you log in. All the examples in this manual assume that the working catalog is $LOCAL.

The discussion of catalogs deals with the following topics:

Using $LOCAL

Using $USER

Determining and Changing the Working Catalog

Catalog Assumed for a File Opened by a COBOL or FORTRAN Program

Referencing Files Owned by Others

Subcatalogs

## Using $LOCAL

$LOCAL is the default working catalog; that is, $LOCAL is the working catalog when you log in to NOS/VE. You can change the working catalog by using the SET_WORKING_CATALOG command, which is discussed later.

If $LOCAL is the working catalog, any file you create in your job is assigned to catalog $LOCAL unless you specify that it be assigned to another catalog. Files in catalog $LOCAL are temporary files; they are deleted when you log out or detach them.

The following example uses two local files. A FORTRAN command specifies compilation of a program on file INFILE and designates a list file FTNOUT. The command appears as follows:

```
/fortran  input=infile  list=ftnout
```

If $LOCAL is the working catalog, $LOCAL does not have to appear in a file reference. If the catalog name appears in the file reference, it appears as follows: $LOCAL.INFILE and $LOCAL.FTNOUT.

## Using $USER

Permanent files are stored in a user catalog associated with your user name. The catalog designation $USER specifies your user catalog, called your master catalog. The format of the file reference for files in your master catalog is:

```
$USER.filename
```

For example, assume that you want to copy a permanent file MYFILE from your catalog to the terminal screen. The command is:

```
/copy_file input=$user.myfile
```

If permanent file MYFILE is a FORTRAN program that you want compiled, you can specify it in a FORTRAN command as follows:

```
/fortran  input=$user.myfile
```

Both catalog and file name are required in this file reference because MYFILE is known only in your permanent file catalog and $LOCAL is the working catalog.

However, if you do not want to type $USER every time you use the file, you can attach the file. The attach process enters the file name (either the local file name you specify or the permanent file name by default) in catalog $LOCAL. Subsequent use of the file requires that you type only its local file name.

The following commands attach permanent file MYFILE and specify the file for FORTRAN compilation:

```
/attach_file  file=$user.myfile

/fortran  input=myfile
```

## Determining and Changing the Working Catalog

The working catalog is the catalog used if you do not specify a catalog on a file reference. To find out which catalog is your working catalog, use the $CATALOG function with the DISPLAY_VALUE (DISV) command. For example:

```
/display_value  value=$catalog
```

NOTE

---

If the catalog is $USER to you, the system shows the name with your family and username and not as $USER.

---

To change the working catalog, use the SET_WORKING_CATALOG (SETWC) command. For example:

```
/set_working_catalog  catalog=$user     Sets the working catalog to your master catalog.

/set_working_catalog  catalog=$local    Sets the working catalog to $LOCAL.

SETWC  any-catalog                      Sets the working catalog to the specified catalog.
```

If you change the working catalog to a catalog other than $LOCAL, the file reference for a file in $LOCAL is as follows:

```
$LOCAL.filename
```

## Catalog Assumed for a File Opened by a FORTRAN or COBOL Program

Catalog $LOCAL is assumed for any file opened by a COBOL or FORTRAN program, even though you set another working catalog. Therefore, the file opened must be attached to $LOCAL if it does not reside there.

For example, if the working catalog is $USER, the following ATTACH_FILE (ATTF) command attaches file TRANFILE in your master catalog to $LOCAL:

```
/attach_file  file=tranfile ..
.. / access_mode=(read  write) ..
.. / share_mode=(read  write)
```

The access modes specify what operations you are allowed to do on a file. The share modes specify what operations other jobs can do to a file while you have it attached. Both have values from the list:

```
READ
APPEND
MODIFY
EXECUTE
SHORTEN
WRITE
ALL
NONE
```

## Referencing Files Owned by Others

To use files of other owners, you need to specify their user name for the catalog name. The format of the file reference for another owner's files (assuming the other owner is in your family) is:

```
.username.filename
```

For example, the file reference to specify user OWNER1's file PROGA as an input file appears as follows:

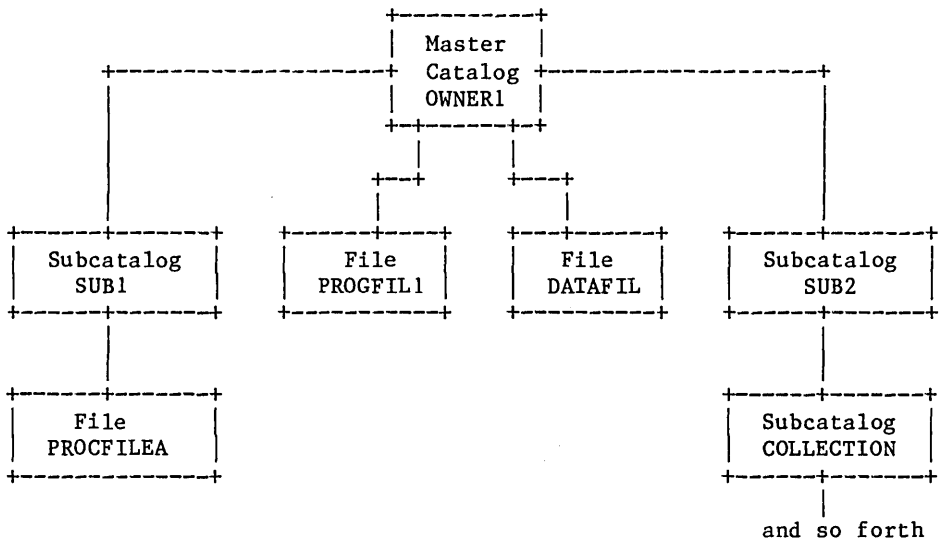
```
/fortran  input=.owner1.proga
```

NOTE

This format is also acceptable for files in your own master catalog. Actually, SCL converts the $USER notation to this form.

## Subcatalogs

Under NOS/VE, catalogs can have subordinate catalogs. The following diagram indicates a possible catalog and file structure for user OWNER1.




A reference to file PROCFILEA must indicate the catalogs associated with the file. In other words, the reference provides a path to the file. For example:

```
/attach_file  file=.owner1.sub1.procfilea
```

Attaches PROCFILEA in subcatalog SUB1 in user catalog OWNER1 of your family.

See the SCL Language Definition Usage manual and the SCL System Interface Usage manual for detailed information about creating and using catalogs.

## File Position

On NOS, you often expect files to be at end-of-information (EOI). NOS/VE, however, usually rewinds files when it opens them, which it does nearly every time a new command operates on a file--not just when an application specifies opening the file. This open position concept is important to understanding file position on NOS/VE. The usual open position is beginning-of-information (BOI); therefore, the file is usually positioned at BOI for every new command operating on it.

Here is an example comparing NOS and NOS/VE. Assume that you want to copy files A, B, and C to file Z. The commands for NOS are:

```
REWIND,A,B,C,Z.
COPY,A,Z.
COPY,B,Z.
COPY,C,Z.
```

The REWIND command positions files A, B, C, and Z at BOI. Because NOS never rewinds file Z, Z is at EOI when the second and third COPY commands are executed.

NOS/VE, however, rewinds all files (A, B, C, and Z) before each copy operation unless you explicitly prevent rewinding by controlling the open position. If you want a file positioned at end-of-information before a copy, you must specify positioning to EOI.

The two ways to specify file position on an open operation are: by specifying file position in a file reference and by using the ATTACH_FILE (ATTF) command. Also, NOS/VE has several file positions that you can specify in these situations. The discussion of file position is divided into the following topics:

Setting File Position With a File Reference

Setting File Position With the ATTACH_FILE Command

Optional File Position Specifiers ($EOI, $BOI, $ASIS)

NOTE

File OUTPUT is automatically positioned at end-of-information on an open operation.

### Setting File Position With a File Reference

The $EOI specification in a file reference positions the file at end-of-information. The following command series uses $EOI:

```
/copy_file  input=a  output=z.$boi
/copy_file  input=b  output=z.$eoi
/copy_file  input=c  output=z.$eoi
```

The second and third copies specify that file Z be positioned at end-of-information before the operation. A period (.) separates the $EOI (or $BOI) specifier from the preceding item in the file reference.

For an example of putting multiple program binaries on a single file, assume that you have separate compilations for a main program and a subprogram but that you want them on the same binary file. The commands are:

```
/fortran  input=mainprog  binary_object=lgo
/fortran  input=subprog  binary_object=lgo.$eoi
```

For another example, assume that you want all your program listings to go to a single listing file. The commands are:

```
/fortran  input=prog1  list=listfil
/fortran  input=prog2  list=listfil.$eoi
```

In each of these situations, the binary file and the listing file would have been rewound without the $EOI specification.

## Setting File Position With the ATTACH_FILE Command

You use an ATTACH_FILE (ATTF) command to set the OPEN_POSITION (OP) attribute to end-of-information ($EOI). This parameter determines file position on an open operation unless that position is otherwise specified.

For example, the following ATTACH_FILE command determines the position for all three copies:

```
/attach_file  file=$user.z  open_position=$eoi  <-- Sets file attribute for file Z.
/copy_file  input=a  output=z  <------------------- Copies file A to file Z.
/copy_file  input=b  output=z  <------------------- Copies file B to file Z.
/copy_file  input=c  output=z  <------------------- Copies file C to file Z.
/copy_file  input=z.$boi  <------------------------ Copies the entire file Z to OUTPUT.
```

## Optional File Position Specifiers

File position can be specified on both the file reference and ATTACH_FILE (ATTF) command. The following positions can be specified:

$BOI  Position to beginning of information.

$ASIS  No file positioning.

$EOI  Position to end-of-information.

These positions designate the file position when the file is opened.

For example, a file reference for the list file for FORTRAN compilation output specifying no file positioning appears as follows:

```
/fortran  input=infile  list=listfile.$asis
```

## Permanent File Cycles

On NOS/VE, a permanent file can have different versions. These versions are known as cycles of a file. A cycle is designated by a cycle number or a cycle reference (such as $HIGH for the latest and $LOW for the earliest cycle). A cycle number can be an integer from 1 through 999.

When you specify a file reference without specifying cycle, the system usually assumes the highest cycle ($HIGH) except when deleting a file, where $LOW is assumed.

For detailed information about using file cycles, see the SCL Language Definition Usage manual.

## Summary of Simple File References

These formats assume that the default working catalog is $LOCAL.

A local file reference has the following format:

```
filename.file position  <-- File position is optional.
```

A file reference for a permanent file has two forms: (1) files in your catalog and (2) files owned by others.

1) $USER.filename.cycle.file position  <------ Cycle and file position are optional.

2) .username.filename.cycle.file position  <-- Cycle and file position are optional.

For more complex file references (that is, specifying family or subcatalogs), see the SCL Language Definition Usage manual.

# Common NOS/VE Commands 4

# Common NOS/VE Commands 4

This chapter discusses commonly used NOS/VE commands from a NOS user's viewpoint. These discussions introduce you to common commands and help you use the commands in simple situations or in migration situations.

First commonly used NOS commands are listed with the corresponding NOS/VE commands. Then the NOS/VE commands are discussed for operations as follows:

Transferring files

Using files

Using file permissions

Miscellaneous operations

Detailed descriptions of the commands are available in the SCL System Interface Usage manual.

## Corresponding NOS-NOS/VE Commands

Commonly used NOS commands are listed together with a NOS/VE command that performs the same function in table 4-1. Knowledge of these equivalent commands is not enough to get your jobs running on NOS/VE. The System Command Language (SCL) for NOS/VE is different from CYBER Control Language (CCL) for NOS. Many useful NOS/VE commands have no NOS equivalents; therefore, understanding and using some of these commands is essential to migrate jobs.

Table 4-1. Corresponding NOS and NOS/VE Commands

| NOS Command | NOS/VE Command and Comments | |
|---|---|---|
| ASSIGN | REQT | REQUEST_TERMINAL. Comparable commands are:<br>NOS: ASSIGN,TT,lfn<br>NOS/VE: REQT lfn |
| ATTACH | ATTF | ATTACH_FILE or an implicit attach when a file is referenced in a command. |
| BATCH | | No equivalent command; no subsystems on NOS/VE. |
| BLANK | | No equivalent command; the operator must blank label the labeled tapes. |
| BRIEF | | No equivalent command. For messages, the SET_MESSAGE_MODE to BRIEF command applies. Many SCL commands have a DISPLAY_OPTION or LIST_OPTION parameter that provides a similar function. |
| CATALOG | | No equivalent command. However, the DISPLAY_OBJECT_LIBRARY command executed within the Object Code Utility (OCU) provides similar operations for object library files. See the SCL Object Code Management Usage manual for information. The SOURCE_CODE_UTILITY subcommands DISPLAY_DECK_LIST and DISPLAY_LIBRARY_LIST provide information about source library files. See the SCL Source Code Management Usage manual for more information. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command and Comments |
|---|---|
| CATLIST | DISC — DISPLAY_CATALOG |
| CATLIST,FN= | DISCE — DISPLAY_CATALOG_ENTRY |
| CHANGE | CREFP — CREATE_FILE_PERMIT<br>DELFP — DELETE_FILE_PERMIT<br>CRECP — CREATE_CATALOG_PERMIT<br>DELCP — DELETE_CATALOG_PERMIT<br>CHACE — CHANGE_CATALOG_ENTRY |
| CHARGE | Logging in gives charge information. |
| CKP | No equivalent command. |
| CLEAR | No equivalent command. To release files, each file must be specified in a DETACH_FILE command. |
| COBOL5 | COBOL — Calls COBOL compiler. Both NOS/VE COBOL and NOS COBOL are based on the ANSI 1974 standard. |
| COMMENT | No equivalent command. Quotation marks specify comments. See chapter 2 for more information on using quotation marks. DISPLAY_MESSAGE (DISM) writes a message to the job log. |
| COMMON | No equivalent command. |
| COPY or COPYEI | COPF — COPY_FILE |
| COPYBF | No equivalent command. |
| COPYBR | No equivalent command. |
| COPYCF | No equivalent command. |
| COPYCR | No equivalent command. |
| COPYL<br>COPYLM | Similar operations with the CREATE_OBJECT_LIBRARY and REPLACE_MODULE command series. See Using Object Libraries in chapter 8 for more information. |
| COPYSBF | PRIF — For printing a file, use the PRINT_FILE command. |
| COPYX | No equivalent command. |
| CSET | No equivalent command. Your terminal always uses the full ASCII character set. |
| CTIME | DISJD — DISPLAY_JOB_DATA |
| DAYFILE | DISL — DISPLAY_LOG<br>CREFC — CREATE_FILE_CONNECTION with standard file $ECHO. For example, the command for a dayfile-like listing on file OUTFILE is:<br>/create_file_connection standard_file=$echo file=outfile<br>File connection terminated by DELFC (DELETE_FILE_CONNECTION) command. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command | and Comments |
|---|---|---|
| DEFINE | CREF | CREATE_FILE or an implicit create when a nonexisting permanent file is referenced in a command. |
| DIAL | | No equivalent command. |
| DISPLAY | DISV | DISPLAY_VALUE |
| DMB<br>DMD<br>DMP | | No dumps on NOS/VE. You can specify the ABORT_FILE parameter on an EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command for information if your application fails. Alternately, you can specify the DISPLAY_MEMORY command if you are using Debug. |
| DMDECS<br>DMPECS | | No equivalent commands. Extended memory is not applicable to virtual memory on NOS/VE. |
| DOCMENT | | No equivalent command. |
| DROP | | TERMINATE_JOB, TERMINATE_TASK, or TERMINATE_PRINT commands. See the SCL System Interface Usage manual for more information. |
| DUP | | Use the text editor (EDIT_FILE) for similar operations. See the File Editor Tutorial/Usage manual for more information. |
| ENQUIRE | | Commands that provide some similar operations are: DISPLAY_JOB_STATUS, DISPLAY_CATALOG, and DISPLAY_JOB_DATA. NOS/VE does not have registers, but the same information can be in variables. To display values of variables, use the DISPLAY_VALUE command. |
| EVICT | | No equivalent command. |
| EXECUTE | | No equivalent command; no subsystems on NOS/VE. |
| EXIT | WHEN | Use for error handling in procedures. |
| EXPLAIN | EXPLAIN | To learn the differences between NOS and NOS/VE online manuals, see the CONTEXT Differences discussion later in this chapter. |
| FCOPY | | No equivalent command. To convert files between NOS and NOS/VE, use GET_FILE and REPLACE_FILE (described following this table). |
| FILE | SETFA<br>ATTF | SET_FILE_ATTRIBUTE<br>ATTACH_FILE |
| FORM | FMU | File Management Utility provides for converting and reformatting files. FMU handles NOS/VE file conversions and NOS to NOS/VE conversions or the reverse. |
| FTN5 | FORTRAN or FTN | Calls FORTRAN compiler. Both NOS/VE FORTRAN and NOS FORTRAN 5 are based on the ANSI 1977 standard. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command and Comments |
|---|---|
| GET | ATTF — ATTACH_FILE Also COPY_FILE (COPF) creates a temporary copy of a permanent file when used as follows:<br>copy_file input=$user.permfile output=temp<br>TEMP is a temporary file. $USER.PERMFILE is file PERMFILE in your master catalog. |
| GTR | No equivalent command. However, to get a procedure out of an object library into a listable file, use the following command series:<br>CREATE_OBJECT_LIBRARY<br>ADD_MODULE MODULE=ocu-deck-to-list ..<br>LIBRARY=library-file-where-deck-resides<br>GENERATE_LIBRARY new-library FORMAT=SCL_PROC<br>QUIT<br>COPY_FILE new-library<br>where new-library is the file that receives the copy of the procedure you wish to list. See the SCL Object Code Management Usage manual for more information. |
| HELPME | DISCI — DISPLAY_COMMAND_INFORMATION lists the parameters of the specified command. For example, for the parameters of ATTACH_FILE command, enter:<br>display_command_information command=attach_file<br>H — The HELP command provides access to the SCL Quick Reference manual, or, if you are in a utility, it gives you any existing help for that utility. If an error just occurred, entering HELP gives you help on that error. Full screen applications typically have a HELP function key. |
| HTIME | No equivalent time. |
| ITEMIZE | DISOL — DISPLAY_OBJECT_LIBRARY displays modules in a library or object file. See chapter 8 for more information. |
| KRONREF | No equivalent command. |
| LABEL | CRE1C, REQMT, CHATLA — CREATE_170_REQUEST references a labeled tape with NOS files. REQUEST_MAGNETIC_TAPE command working with the CHANGE_TAPE_LABEL_ATTRIBUTES command. |
| LBC | No equivalent command. |
| LDI | SUBJ — SUBMIT_JOB |
| LENGTH | DISFA — DISPLAY_FILE_ATTRIBUTES |
| LGO | LGO — Default file to which compilers write object code. However, command EXET (EXECUTE_TASK) is also available. |
| LIB | No equivalent command. |
| LIMITS | The following command series displays your validation information:<br>ADMINISTER_USER<br>DISPLAY_USER<br>QUIT |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command | and Comments |
|---|---|---|
| LIST | COPF | COPY_FILE |
| LISTLB | DISTLA | DISPLAY_TAPE_LABEL_ATTRIBUTES lists much of the same tape label information as does LISTLB. |
| LOC | | No equivalent command. |
| LOCK | ATTF | Use the ACCESS_MODE parameter in the ATTACH_FILE command. |
| LO72 | | No equivalent command. |
| MACHINE | | No equivalent command. |
| MERGE | MERGE | Initiates the merge operation of the Sort/Merge Utility. |
| MFL | | No equivalent command. This is not necessary on NOS/VE; however, the operation can be simulated by: CHANGE_JOB_ATTRIBUTE MAXIMUM_WORKING_SET=integer |
| MFLINK | | See commands described under the Permanent File Transfer Facility in chapter 11. |
| MODE | SETPA | SET_PROGRAM_ATTRIBUTES can specify the handling for arithmetic and/or BDP (Business Data Processing) conditions. |
| MOVE | | You can perform this operation by using editor commands. See the File Editor Tutorial/Usage manual for more information. |
| NEW | | No equivalent command. Primary files do not exist on NOS/VE. |
| NOEXIT | | No equivalent command. Use WHEN command for error handling in procedures. |
| NORERUN | | No equivalent command. |
| NORMAL | | No equivalent command. Your terminal always uses the full ASCII character set. |
| NOSORT | | No equivalent command. |
| NOTE | PUTL | PUT_LINE |
| | COLT | COLLECT_TEXT |
| NULL | | No equivalent command; no subsystems on NOS/VE. |
| OFFSW | SETSS | SET_SENSE_SWITCH; see the SCL System Interface Usage manual for information. |
| OLD | | No equivalent command. |
| ONSW | SETSS | SET_SENSE_SWITCH; see the SCL System Interface Usage manual for information. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command and Comments |
|---|---|
| OPLEDIT | Use the Source Code Utility (SCU) commands DELETE_MODIFICATION, DELETE_DECK, and EXTRACT_MODIFICATION. See the SCL Source Code Management Usage manual for more information. |
| OUT PRIF | PRINT_FILE |
| PACK | No equivalent command. |
| PACKNAM | No equivalent command. |
| PASSWOR SETPW | SET_PASSWORD; see the SCL System Interface Usage manual for information. |
| PAUSE | No equivalent command. |
| PERMIT CREFP<br>DELFP<br>CRECP<br>DELCP | CREATE_FILE_PERMIT<br>DELETE_FILE_PERMIT<br>CREATE_CATALOG_PERMIT<br>DELETE_CATALOG_PERMIT |
| PRIMARY | No equivalent command. |
| PROTECT | No equivalent command. |
| PURGALL | No equivalent command. To delete a catalog, specify:<br>BACKUP_PERMANENT_FILE BACKUP_FILE=$NULL LIST=$OUTPUT<br>DELETE_CATALOG_CONTENT CATALOG=catalog name<br>QUIT<br>To delete all files except the high cycles, specify:<br>BACKUP_PERMANENT_FILE BACKUP_FILE=$NULL LIST=$OUTPUT<br>EXCLUDE_HIGHEST_CYCLES NUMBER_OF_CYCLES=1<br>DELETE_CATALOG_CONTENT CATALOG=catalog name<br>QUIT |
| PURGE DELF | DELETE_FILE |
| QGET | No equivalent command. |
| RBR | No equivalent command. |
| RECOVER ATTJ | ATTACH_JOB puts your job in a pause break state. To resume processing, specify either TERMINATE_COMMAND (TERC) or RESUME_COMMAND (RESC). |
| REPLACE COPF | COPY_FILE can rewrite a permanent file with a temporary modified copy when used as follows:<br>/copy_file input=temp output=$user.permfile<br>This example assumes TEMP is a modified temporary copy of file PERMFILE which resides in your master catalog. |
| REQUEST REQMT | REQUEST_MAGNETIC_TAPE to request NOS/VE tape; tape assignment does not occur until the file is actually used. |
| CRE1R | CREATE_170_REQUEST to request NOS tape; tape assignment does not occur until the file is actually used. |
| RERUN | No equivalent command. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command | and Comments |
|---|---|---|
| RESEQ | | In general, there is no equivalent command. BASIC's RESEQUENCE utility is comparable to RESEQ. |
| RESOURC | RESR<br>RELR | RESERVE_RESOURCE<br>RELEASE_RESOURCE |
| RESTART | | No equivalent command. |
| RETURN | DETF | DETACH_FILE |
| RFL | | No equivalent command. The operation is not needed on NOS/VE. |
| ROLLOUT | WAIT | Suspend current task. |
| RUN | | No equivalent command. |
| SAVE | COPF | COPY_FILE can create a permanent file from a local file. For example:<br>/copy_file input=temp output=$user.permfile<br>Local file TEMP is copied and creates permanent file PERMFILE in your master catalog. |
| SCOPY | | No equivalent command. |
| SETASL | CHAJL | CHANGE_JOB_LIMIT |
| SETCORE | | Specify the PRESET_VALUE attribute on either of the following commands: EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES. See the SCL Object Code Management Usage manual for details. |
| SETFS | | No equivalent command. |
| SETJOB | SUBJ<br>LOGIN | JOB_NAME parameter of the LOGIN and SUBMIT_JOB (SUBJ) commands. This applies only for batch jobs. |
| SETJSL | CHAJL | CHANGE_JOB_LIMIT |
| SETPR | CHAJA | DISPATCHING_PRIORITY parameter of CHANGE_JOB_ATTRIBUTES command. |
| SETTL | CHAJL | CHANGE_JOB_LIMIT |
| SKIP | | IF ...,THEN sequence. |
| SKIPEI | | Use .$EOI on a file reference. See the file position discussion in chapter 3 for more information. |
| SKIPF | | No equivalent command. |
| SKIPFB | | No equivalent command. |
| SKIPR | | No equivalent command. |
| SORT5 | SORT | Initiates the sort operation of the Sort/Merge Utility. |
| STIME | | No equivalent command. |

(Continued on next page)

Table 4-1. Corresponding NOS and NOS/VE Commands

(Continued from previous page)

| NOS Command | NOS/VE Command and Comments | |
|---|---|---|
| SUBMIT | SUBJ | SUBMIT_JOB |
| SUMMARY | | Commands that provide some similar operations are: DISPLAY_JOB_STATUS, DISPLAY_CATALOG, and DISPLAY_JOB_DATA. NOS/VE does not have registers, but the same information can be in variables. To display values of variables, use DISPLAY_VALUE or PUT_LINE commands. |
| SWITCH | SETSS | SET_SENSE_SWITCH |
| TCOPY | | No equivalent command. |
| TDUMP | DISF | DISPLAY_FILE |
| TEXT | COLT | COLLECT_TEXT or use an editor. |
| TIMEOUT | | No equivalent command. |
| TRMDEF | SETTA | SET_TERMINAL_ATTRIBUTE |
| UNLOAD | DETF | DETACH_FILE |
| UNLOCK | | ACCESS_MODE parameter in commands as follows: for local files, use the SET_FILE_ATTRIBUTE command; for permanent files, use the ATTACH_FILE command. You would first detach your permanent file and then reattach it with a different access mode. |
| UPROC | ADMU | CHANGE_USER and SET_USER_PROLOG subcommands of ADMINISTER_USER command. |
| USECPU | SETMO | SET_MULTIPROCESSING_OPTION |
| USER | LOGIN<br>SETLA | Applies only to batch processing.<br>SET_LINK_ATTRIBUTE specifies user name to gain access to NOS permanent files or to execute NOS commands through EXECUTE_INTERSTATE_COMMAND. |
| VERIFY | COMF | COMPARE_FILE |
| VFYLIB | COMOL | COMPARE_OBJECT_LIBRARY |
| WBR | | No equivalent command. |
| WHATJSN | | No equivalent command. |
| WRITE | | Use the editor. See the File Editor Tutorial/Usage manual. |
| WRITEF | | No equivalent command. However, in the Source Code Utility, the WEOP directive can write end-of-partition marks to partition a file. You can also use the editor. See the File Editor Tutorial/Usage manual. |
| WRITEN | | No equivalent command. |
| WRITER | | No equivalent command. |
| X | | No equivalent command. |
| XMODEM | XMODEM | TRANSFER_FILE_XMODEM |

## Commands for Transferring Files

Transferring text files is handled easily with the GET_FILE (GETF) and REPLACE_FILE (REPF) commands. GETF converts and transfers files from NOS to NOS/VE (GETF); REPF does the reverse operation.

Do not use GET_FILE or REPLACE_FILE with a file that is not a text file. The converted file might contain incorrectly formatted items. Processing the file can yield unpredictable results, including program aborts in NOS/VE due to hardware errors.

Be sure that GET_FILE and REPLACE_FILE refer to the files properly. If there are any accounting information (family name, user name, etc.) differences between NOS and NOS/VE, you might need to use the SET_LINK_ATTRIBUTE command. The SCL System Interface Usage manual discusses this command.

The text files can be either data or program source code. Files transferred with these commands are typically files created with a text editor, COBOL DISPLAY statement, or FORTRAN PRINT or formatted WRITE statements.

### GET_FILE (GETF)

The NOS/VE command GET_FILE transfers a copy of a NOS permanent file (direct or indirect access) to NOS/VE and converts it to NOS/VE format. The file can be a text or binary file; however, this guide discusses only text files. The text file must have Z-type (zero-byte) records. (Files created by text editors and by the COBOL DISPLAY statement and the FORTRAN PRINT and formatted WRITE statements have Z-type records.)

GETF performs data conversion on the file transfer. The default conversion is NOS ASCII (in 6/12-bit display code format) to NOS/VE 7-bit ASCII code. NOS generates 6/12-bit display code when you specify ASCII mode; otherwise, NOS generates 6-bit display code (NORMAL mode). To transfer files in 6-bit display code, specify the parameter DATA_CONVERSION=D64 (or DC=D64).

Simple examples:

```
/get_file  to=myfile
```

Transfers NOS file MYFILE (in 6/12-bit display code) to NOS/VE as local file MYFILE.

```
/get_file  to=joba  data_conversion=d64
```

Transfers NOS file JOBA (in 6-bit display code) to NOS/VE with the local file name JOBA.

```
/get_file  to=$user.afile  from=myfile
```

Transfers NOS file MYFILE (in 6/12-bit display code) to NOS/VE with permanent file name AFILE. (For information about $USER, see Using $USER in chapter 3.)

#### GET_FILE (GETF) Format

Below is a commonly used form of the GET_FILE command:

```
GET_FILE  T=nos/ve file  F=nos file  DC=option  U=user  PW=password
```

Parameters:

T (TO) Designates the NOS/VE file to which the file is copied. Required parameter. Default characteristics are assumed about the file unless you specify the characteristics in a SET_FILE_ATTRIBUTES command. See the File Attributes discussion in chapter 10, File Interface Introduction, for additional information.

F (FROM) Designates the NOS file to be copied. The default is the same name designated for the TO parameter.

DC (DATA_CONVERSION) Designates conversion to be done on the copy. The text conversion options are:

- A6 Each NOS ASCII character (in 6/12-bit display code) is converted to a 7-bit ASCII character. This is the default option.
- A8 Each NOS ASCII character (in 12-bit ASCII format) is converted to a 7-bit ASCII character.
- D63 Each NOS character in 6-bit display code (63-character set) is converted to a 7-bit ASCII character.
- D64 Each NOS character in 6-bit display code (64-character set) is converted to a 7-bit ASCII character.

Parameters exist for converting binary data. See the SCL System Interface Usage manual for more information.

U (USER) Specifies user name of the owner of the NOS file. Defaults to the user identification you used to log in.

PW (PASSWORD) Specifies the NOS file password needed to access the file. The password is required if you do not own the file and the file has a password.

## REPLACE_FILE (REPF)

REPLACE_FILE transfers a file from NOS/VE to a NOS direct or indirect access permanent file. If the file exists, REPLACE_FILE rewrites the file. If the file does not exist, REPLACE_FILE performs a DEFINE operation. REPLACE_FILE can handle either a text or binary file; however, this manual discusses only text files.

The DATA_CONVERSION parameter indicates data conversion. REPLACE_FILE assumes ASCII characters (in 6/12-bit display code format) for the NOS files. To convert a NOS/VE text file to 6-bit display code for NOS, specify DATA_CONVERSION=D64. Examples:

```
/replace_file  from=filea
```

Transfers NOS/VE file FILEA to NOS file FILEA with ASCII characters in 6/12-bit display code.

```
/replace_file  from=nvefile  to=nosfile
```

Transfers NOS/VE file NVEFILE to NOS file NOSFILE with ASCII characters in 6/12-bit display code.

```
/replace_file  from=jobfile  data_conversion=d64
```

Transfers local NOS/VE file JOBFILE to NOS, with characters in 6-bit display code.

### REPLACE_FILE (REPF) Format

Below is a commonly used form of the REPLACE_FILE command:

```
REPLACE_FILE  F=nos/ve file  T=nos file  DC=option  U=user  PW=password
```

Parameters:

F (FROM) Designates NOS/VE file to be copied. (Required.)

T (TO) Designates NOS indirect or direct access file to which the file is copied. Defaults to file name on the FROM parameter. The text files are written with Z-type records.

DC (DATA_CONVERSION) Designates conversion to be done on the copy. The NOS text files are writen with Z type records.

- A6 NOS/VE 7-bit ASCII characters are converted to NOS ASCII in 6/12-bit display code format. This is the default option.
- A8 NOS/VE 7-bit ASCII characters are converted to NOS 12-bit ASCII format.
- D63 NOS/VE 7-bit ASCII characters are converted to NOS 6-bit display code (63-character set).
- D64 NOS/VE 7-bit ASCII characters are converted to NOS 6-bit display code (64-character set).

U (USER) Specifies user name of the owner of the NOS file. Defaults to your user identification you used to log in.

PW (PASSWORD) Specifies the NOS file password needed to access the file. The password is required if you do not own the file and the file has a password.

## Commands for Using Files

The EDIT_CATALOG (EDIC) is a full screen utility that makes it easy to do a number of common operations on files. This command is described in the SCL System Interface Usage manual.

The following discussion describes commands to create, delete, attach, detach, and copy files, as well as commands to display information about files. Also, creating or attaching a file without an explicit command is discussed.

### CREATE_FILE (CREF)

CREATE_FILE creates a new empty permanent file. The command also indicates the time period to retain the file and can specify a password and other information.

CREATE_FILE is equivalent to the DEFINE command on NOS. On NOS/VE files are not designated as direct or indirect access files. Consider the files as being direct access.

Simple example:

```
/create_file  file=$user.data_file
```

This command creates a permanent file DATA_FILE in your user catalog and attaches the file, making it available to your job by local file name DATA_FILE. No password is specified. No retention period is specified; therefore, an indefinite period is assumed.

The CREATE_FILE command is not necessary to create all new permanent files. An implicit create can occur by using a file reference for a file that does not exist.

### CREATE_FILE (CREF) Format

Below is a commonly used form of the CREATE_FILE command:

```
CREATE_FILE  F=permfile  TFN=temporary file name  PW=password
```

Parameters:

F (FILE) File reference; for more information see the discussion of file reference in chapter 3, Overview of the Permanent File Mechanism.

LFN (LOCAL_FILE_NAME) Local file name for the file; the default is the permanent file name.

PW (PASSWORD) Specifies password that must be used for all subsequent access to any cycle of the file.

## Implicit Create

An implicit create occurs when you specify a file reference for a permanent file when no permanent file exists.

Example:

```
/get_file  to=$user.myfile
```

This command transfers NOS permanent file MYFILE to NOS/VE as permanent file MYFILE under your user name and adds it to your permanent file catalog.

## DELETE_FILE (DELF)

DELETE_FILE deletes a permanent file. If the file has cycles, DELETE_FILE deletes one file cycle (the earliest; that is, lowest cycle by default).

The DELETE_FILE command is equivalent to the PURGE command on NOS.

Simple example:

```
/delete_file  file=$user.myfile
```

This command deletes permanent file MYFILE in your user catalog if MYFILE has no cycles. If MYFILE has cycles, the command deletes the earliest cycle of the file.

Notice that there is only one letter difference between the abbreviations for the DELETE_FILE and the DETACH_FILE commands: DELF versus DETF. The one letter can have a powerful effect. If you have a permanent file attached to your job and you specify DELF tfn (tfn is the temporary file name for the permanent file), you delete your permanent file as well as your temporary file.

For example:

```
/attach_file  file=$user.myfile  local_file_name=infile
```

Gives permanent file MYFILE the temporary file name INFILE.

```
/delete_file  file=infile
```

Deletes $USER.MYFILE as well as the temporary file INFILE.

Because of the only one character difference in the abbreviations, be careful deleting and detaching local files when you have permanent files attached.

### DELETE_FILE (DELF) Format

Below is a commonly used form of the DELETE_FILE command:

```
DELETE_FILE  F=file  PW=password
```

Parameters:

F (FILE) Identifies the permanent file. Omission of cycle causes $LOW (the oldest) to be used.

PW (PASSWORD) Specifies the file password.

## ATTACH_FILE (ATTF)

ATTACH_FILE attaches a permanent file to a job. In other words, ATTACH_FILE gives the file an alias by which it is known in the working catalog ($LOCAL is the default working catalog). Permanent files used for input and output by COBOL or FORTRAN programs must be attached if the program specifies the file names. The files must be attached because a file reference, which indicates a permanent file catalog ($USER for example), cannot appear in a COBOL or FORTRAN statement.

Once you have attached a file explicitly, any command that depends on the path name for a file is affected. The command uses the temporary path name rather than the path specified in the command. For example:

```
/attach_file   file=$user.x  local_file_name=y
/replace_file  from=$user.x
```

results in file Y, not file $USER.X, being transferred to NOS.

The ATTACH_FILE command is equivalent to the ATTACH command and similar to GET command on NOS.

Simple example:

```
ATTF $USER.OUTDATA AM=(READ WRITE)     Attaches permanent file OUTDATA in your user catalog
 .                                     with read and write permission.
 .

      OPEN(2,FILE='OUTDATA')           Subsequent FORTRAN open operation using OUTDATA.
```

### ATTACH_FILE (ATTF) Format

Below is a commonly used form of the ATTACH_FILE command:

```
ATTACH_FILE  F=file  LFN=temporary file name  PW=password  AM=access mode  SM=share mode
```

Parameters:

F (FILE) Identifies the permanent file. If cycles exist, $HIGH is assumed.

LFN (LOCAL_FILE_NAME) Specifies the local file name to be used for the file within the job. The default for lfn is the permanent file name. Avoid using this parameter. Its use makes it difficult for you to track the status of the temporary file.

PW (PASSWORD) Specifies an optional file password.

AM (ACCESS_MODE) Specifies how the file is to be used within the job. Values are: READ, APPEND, MODIFY, EXECUTE, SHORTEN, WRITE, or ALL. (WRITE is equivalent to specifying APPEND, MODIFY, and SHORTEN.) The default values are READ and EXECUTE modes.

SM (SHARE_MODE) Specifies how the file can be shared with other users while attached to your job. (See the SCL System Interface Usage manual for details.)

## Implicit Attach

An ATTACH_FILE command is not necessary to make a permanent file available to your job. An implicit attach occurs when you use a file reference for a permanent file in an SCL command. The implicit attach automatically attaches the file in the appropriate access mode for use by your job. Implicitly attached files are automatically detached when a program or command ends.

Examples:

```
/copy_file  input=outfile  output=$user.myfile
```

Copies the contents of local file OUTFILE to your permanent file MYFILE.

```
/execute_task  file=$user.binfile
```

Executes permanent binary file BINFILE.

Avoid using an implicit attach in a batch job if the file could be busy. An implicit attach does not wait for the file if the file is busy; the job aborts. Use the ATTACH_FILE command to wait for the file to be available.

NOTE

When NOS/VE accesses a file, it positions the file to $BOI unless you take measures to prevent the automatic positioning. See the File Position discussion in chapter 3, Permanent File Overview, for more information about file positioning.

# DETACH_FILE (DETF)

DETACH_FILE detaches one or more files from a job. A temporary file is deleted. A permanent file no longer has the local alias name.

The DETACH_FILE command is equivalent to the RETURN command on NOS.

Examples:

```
/detach_file  file=list
```

Detaches local file LIST.

```
/detach_file  files=(list  errout  temp)
```

Detaches the specified local files.

### DETACH_FILE (DETF) Format

Below is a commonly used form of the DETACH_FILE command:

```
DETACH_FILE  F=files
```

Parameters:

F (FILE or FILES) Specifies a local file or list of local files to be detached. A list of files must be enclosed in parentheses.

## COPY_FILE (COPF)

COPY_FILE copies data from one file to another. It copies the file to end-of-information.

The COPY_FILE command is equivalent to the COPYEI command on NOS except that it has no verify option.

Simple examples:

```
/copy_file  input=filea  output=fileb
```

Copies the contents of local file FILEA to local file FILEB.

```
/copy_file  input=$user.myfile  output=newfile
```

Copies the contents of permanent file MYFILE to local file NEWFILE.

Usually, both input and output files are rewound before the copy. See the File Position discussion in chapter 3, Permanent File Overview, for more information about controlling file positioning.

### COPY_FILE (COPF) Format

Below is a commonly used form of the COPY_FILE command:

```
COPY_FILE  I=input file  O=output file
```

Parameters:

I (INPUT) Specifies the file from which data is copied. The default file is $INPUT. Data is copied from the open position until end-of-information is reached.

O (OUTPUT) Specifies the file to which data is copied. The default file is $OUTPUT.

If the copy creates a file, the new file generally inherits the attributes of the old; however, exceptions exist. See the SCL System Interface Usage manual for details about file compatibilities of the input and output files.

## DISPLAY_CATALOG (DISC)

DISPLAY_CATALOG displays information about the files in a catalog (or catalogs in a catalog).

The DISPLAY_CATALOG command is equivalent to the CATLIST command on NOS.

**Examples:**

```
/display_catalog
```

Displays a listing of the files in the working catalog, usually catalog $LOCAL.

```
/display_catalog  catalog=$user
```

Displays a list of the files in your master catalog.

```
/display_catalog  catalog=$user  display_options=permits
```

Displays access control permissions.

### DISPLAY_CATALOG (DISC) Format

Below is a commonly used form of the DISPLAY_CATALOG command:

```
DISPLAY_CATALOG  C=catalog  DO=option
```

Parameters:

C (CATALOG) Specifies the catalog from which information is being displayed.

DO (DISPLAY_OPTION or DISPLAY_OPTIONS) Specifies the type of display being requested. Values include:

| | |
|---|---|
| ID | Display of the permanent file or catalog names in the catalog together with its type (file or catalog). This is the default value (DO=ID). |
| FILE | Summary description of files in the catalog including the number of cycles. |
| PERMITS | Previously specified permission for file access. For more infor- mation see the discussion of CREATE_FILE_PERMIT in this chapter. |

## Commands for Using File Permissions

To begin using file permissions, you should be familiar with the following commands:

CREATE_FILE_PERMIT

DELETE_FILE_PERMIT

CHANGE_CATALOG_ENTRY

DISPLAY_CATALOG_ENTRY

### CREATE_FILE_PERMIT (CREFP)

This command establishes or modifies an access control entry for a specific permanent file. This command is comparable to the PERMIT command on NOS.

Another command, CREATE_CATALOG_PERMIT, provides for the same access control as this command does, only at the catalog level. Other associated commands briefly discussed in this manual are:

| | |
|---|---|
| DELETE_FILE_PERMIT | Deletes specified file permissions. |
| CHANGE_CATALOG_ENTRY | Changes established permissions. |
| DISPLAY_CATALOG_ENTRY | Shows established permissions. |

You should be aware of the CREATE_FILE_PERMIT command if you want to protect your files by granting limited access or if you are having trouble accessing a file that you know exists.

Several access control entries can apply to a user requesting access. In this case, the access control entry associated with the smallest group is used. For example, if a file has an access control entry for a user and another access control entry for a user's family, then the access control entry for the user is applied.

If several access control entries specify the same group of users, then the entry referring to the lowest level in the master catalog, subcatalog, and file hierarchy is used. For example, if an account has an access control entry for a file and another access control entry for a catalog containing that file, then the former access control entry is used.

For more information about this command, see the SCL System Interface Usage manual.

### CREATE_FILE_PERMIT (CREFP) Examples

The following command gives user BOND007 permission to only read or append your permanent file SUPER_RESOURCE:

```
/create_file_permit  file=$user.super_resource ..
.. /user=bond007 ..
.. /access_mode=(read append)
```

This command makes your permanent file SUPER_RESOURCE available to anyone (PUBLIC) to read and append:

```
/create_file_permit  file=$user.super_resource ..
.. /group=public ..
.. /access_mode=(read append) ..
.. /share_mode=(read append)
```

Two parameters control access situations. ACCESS_MODE (AM) specifies how the file can be used. SHARE_MODE (SM) specifies access when several users are accessing (sharing) the file.

### CREATE_FILE_PERMIT (CREFP) Format

A commonly used format of the CREATE_FILE_PERMIT command is:

```
CREATE_FILE_PERMIT or
CREFP
    FILE=file
    GROUP=keyword
    FAMILY_NAME=name
    USER=name
    ACCOUNT=name
    PROJECT=name
    ACCESS_MODE=list of keyword
    SHARE_MODE=list of keyword
```

#### FILE or F

Required. Specifies the permanent file for which the access control entry is being established.

### GROUP or G

Optional. Specifies the kind of user group that the access permission applies to. Its default is USER. The GROUP parameter is similar to the file permit category (CT) in NOS. Valid selections are:

| | |
|---|---|
| PUBLIC | The permit entry applies to all users regardless of family, account, project, or user identifications. This selection is comparable to the NOS PUBLIC file permit category. |
| FAMILY | The permit entry applies to all users in the specified family. |
| ACCOUNT | The permit entry applies to all users associated with the specified family and account identifications. |
| PROJECT | The permit entry applies to all users associated with the specified family, account, and project identifications. |
| USER | The permit entry applies to the specified family and user identifications. |
| USER_ACCOUNT | The permit entry applies to the specified family, account, and user identifications. |
| MEMBER | The permit entry applies to the specified family, account, project, and user identifications. |

The FAMILY_NAME, USER, ACCOUNT, and PROJECT parameters identify the specific user group that the access permission is for. If you omit one of these identifying parameters when the GROUP selection indicates that it is applicable, then the identifier value associated with the requesting job is used. For example, suppose that the GROUP selection is PROJECT, and that you omit the FAMILY_NAME, ACCOUNT, and PROJECT parameters. Then the access permission applies to all users having the family, account, and project associated with the requesting job.

If one of these identifying parameters is specified when the GROUP selection indicates that it is not applicable, an abnormal status is returned.

### FAMILY_NAME or FN

Optional. Specifies the family name to be permitted access.

### USER or U

Optional. Specifies the user name to be permitted access. Its use here is similar to that of $username_i$ in the NOS PERMIT command.

### ACCOUNT or A

Optional. Specifies the account name to be permitted access.

### PROJECT or P

Optional. Specifies the project name to be permitted access.

### ACCESS_MODE or ACCESS_MODES or AM

Optional. Specifies the kind of file access the user group is allowed. The default is READ and EXECUTE. The ACCESS_MODE parameter is similar to the user permission mode (M) for semiprivate and public files in NOS. The selections are:

| | |
|---|---|
| READ | Reading the file is allowed. |
| APPEND | Appending information to the end of the file is allowed. |
| MODIFY | Altering data within the existing file is allowed. |
| SHORTEN | Deleting information from the end of the file is allowed. |
| WRITE | Access is permitted for SHORTEN, MODIFY, and APPEND. |
| EXECUTE | Executing object code or an SCL procedure in the file is allowed. |
| ALL | Access is permitted for READ, APPEND, MODIFY, SHORTEN, WRITE, and EXECUTE. |
| CONTROL | Deleting a file or changing its file identity (file name, cycle number, password, log selection, retention, and charge attributes) and/or file attributes is allowed. |
| CYCLE | Adding new file cycles is allowed. |
| NONE | File access is specifically prohibited. |

### SHARE_MODE or SHARE_MODES or SM

Optional. Specifies the share modes required of jobs that are allowed access to your file. The default is READ and EXECUTE. The share modes are:

| | |
|---|---|
| READ | Sharing the file for READ access. |
| APPEND | Sharing the file for APPEND access. |
| MODIFY | Sharing the file for MODIFY access. |
| SHORTEN | Sharing the file for SHORTEN access. |
| WRITE | Sharing the file for SHORTEN, MODIFY, and APPEND access. |
| EXECUTE | Sharing the file for EXECUTE access. |
| ALL | Sharing the file for READ, APPEND, MODIFY, SHORTEN, WRITE, and EXECUTE access. |
| NONE | No sharing requirements are imposed on jobs permitted to access your file. They can select exclusive access or any of the READ, MODIFY, SHORTEN, WRITE, and EXECUTE share modes. |

## DELETE_FILE_PERMIT (DELFP)

Deletes an access control permission that was established for a permanent file with the CREATE_FILE_PERMIT (CREFP) command. Corresponds to the PERMIT command on NOS.

Examples show a CREATE_FILE_PERMIT command that creates a file permit for a file in your master catalog and the DELETE_FILE_PERMIT command that cancels it.

```
/create_file_permit  file=$user.super_resource ..
.. /user=bond007 ..
.. /access_mode=(read  append)

/delete_file_permit  file=$user.super_resource ..
.. /user=bond007

/create_file_permit  file=$user.super_resource ..
.. /group=public ..
.. /access_mode=(read  append) ..
.. /share_mode (read  append)

/delete_file_permit  file=$user.super_resource ..
.. /group=public
```

See the SCL System Interface Usage manual for more information.

The command DELETE_CATALOG_PERMIT provides the same function as DELETE_FILE_PERMIT, only for a catalog.

## CHANGE_CATALOG_ENTRY (CHACE)

The CHANGE_CATALOG_ENTRY command alters the file name, cycle, password, log selection, retention period, charge identification, and damage condition attributes associated with a file. This command corresponds to the CHANGE command on NOS.

You can issue the CHANGE_CATALOG_ENTRY command only if you have CONTROL permission (see the CREATE_FILE_PERMIT discussion earlier in this chapter for an explanation of CONTROL permission). If the file has a password associated with it, you must also specify the password.

For example, the following statement changes the name, password, and cycle number for cycle number 1 of file DATA_FILE_1 in subcatalog CATALOG_1 in the master catalog:

```
/change_catalog_entry  file=$user.catalog_1.data_file_1.1 ..
.. /new_file_name=data_file_0 ..
.. /new_cycle=87 ..
.. /new_password=new_data_0_pw
```

The name of the file is changed from DATA_FILE_1 to DATA_FILE_0 for all cycles in the file, the cycle number for cycle 1 is changed to 87 (other cycle numbers remain unchanged), and the password for the file is changed from null to NEW_DATA_0_PW.

For more information about this command, see the SCL System Interface Usage manual.

Below is a commonly used form of the CHANGE_CATALOG_ENTRY command:

```
CHANGE_CATALOG_ENTRY or
CHACE
    FILE=file
    PASSWORD=name or keyword
    NEW_FILE_NAME=name
    NEW_CYCLE=integer
    NEW_PASSWORD=name or keyword
    NEW_ACCOUNT_PROJECT=boolean
    DELETE_DAMAGE_CONDITION=keyword
```

### FILE or F

Required. Specifies the permanent file whose attributes are to be changed.

### PASSWORD or PW

Optional. Specifies the current file password. It must match the file password stored with the catalog entry or an abnormal status is returned. The keyword NONE indicates that no password has been specified for the file. Omission of the PASSWORD parameter causes NONE to be used.

### NEW_FILE_NAME or NFN

Optional. Specifies the new file name to be associated with the file. This parameter is like the new file name (nfn) parameter in the NOS CHANGE command. All existing cycles of the file will have the new name. If the new name already exists in the catalog, an abnormal status is returned. Omission of this parameter causes the current name to be retained.

### NEW_CYCLE or NC

Optional. Specifies the new cycle number (from 1 through 999) to be associated with a file cycle.

### NEW_PASSWORD or NPW

Optional. Specifies the new password to be associated with all cycles of the file. This parameter is like the password (PW) parameter in the NOS CHANGE command. The keyword NONE indicates that no password is to be associated with the file. Omission of the PASSWORD parameter causes the current password to be retained.

### NEW_ACCOUNT_PROJECT or NAP

Optional. Specifies whether new account and project identifiers are to be established for the file. These identifiers are used for billing purposes and apply to all cycles of the file. NOS/VE would take the new identifiers from the project and account identification associated with the requesting job. The NEW_ACCOUNT_PROJECT parameter is like the CP parameter in the NOS CHANGE command. Omission of this parameter causes the current account and project identifiers associated with the file to be retained.

### DELETE_DAMAGE_CONDITION or DDC

Optional. Specifies the damage condition information to be deleted from the catalog registration of the file cycle. This parameter enables you to eliminate the exception condition reported when the cycle is attached or opened. It is similar to the CE parameter in the NOS CHANGE command.

The system sets the damage state of the file cycle to indicate the damage condition and reports this condition when the file is attached or opened. The DELETE_DAMAGE_CONDITION parameter enables you to acknowledge that you understand that the damage has occurred and to complete command processing. Once the outstanding damage condition has been deleted, you can attach the file. If the file cycle is not in the specified condition, this command returns an abnormal status, and the other parameters in the command are ignored.

The only allowable keyword is RESPF_MODIFICATION_MISMATCH (RMM). This keyword indicates that a file cycle was restored even though the last modification date and time were different from the online cycle. If you omit this parameter and a cycle damage condition is present, the system will display a diagnostic message.

### DISPLAY_CATALOG_ENTRY (DISCE)

This command displays catalog information about a permanent file--file use and file access control. If the file belongs to another user, the display is granted only if the requesting user is permitted access to the file. This command corresponds to the CATLIST command on NOS.

Example:

```
/display_catalog_entry  file=$user.super_resource  display_options=permits
```

Displays at the terminal the file permissions established for your permanent file SUPER_RESOURCE.

Notice that ALL is not an option for the DISPLAY_OPTIONS parameter. The options need to be specified.

Below is a commonly used form of the DISPLAY_CATALOG_ENTRY command:

```
DISPLAY_CATALOG_ENTRY  F=file  DO=option
```

Parameters:

F (FILE) Specifies a file reference identifying the file that is the subject of the query.

DO (DISPLAY_OPTION or OPTIONS) Specifies the type of information. The options include:

| | |
|---|---|
| LOG | Displays information about file use. |
| PERMITS | Displays information about established access control permits. |
| DESCRIPTOR | Displays information about the permanent file and file cycles. Default value. |

## Using File Attributes

A file attribute is a characteristic of the file. File attributes describe the structure of the file and define processing limitations for the file.

The three commands for using file attributes are:

SET_FILE_ATTRIBUTE (SETFA)

CHANGE_FILE_ATTRIBUTE (CHAFA)

DISPLAY_FILE_ATTRIBUTE (DISFA)

### SET_FILE_ATTRIBUTE (SETFA)

SET_FILE_ATTRIBUTES sets the attributes of a file. The file attributes are used to manage the content and processing of the file.

Examples:

The command:

```
/set_file_attributes  file=$user.result ..
.. /file_organization=sequential ..
.. /maximum_record_length=80
```

sets the file_organization (FO) and maximum_record_length (MAXRL) for permanent sequential file RESULT.

The command:

```
/set_file_attributes  file=ind_seq_file ..
.. /file_organization=indexed_sequential ..
.. /maximum_record_length=65 ..
.. /key_length=10 ..
.. /key_type=uncollated ..
.. /record_type=fixed ..
.. /embedded_key=true
```

sets the file_organization, maximum_record_length, key_length (KL), key_type (KT), record_type (RT), and embedded_key (EK) for the temporary indexed sequential file IND_SEQ_FILE.

## SET_FILE_ATTRIBUTE (SETFA) Format

```
SETFA F=file    list of file attributes
```

Parameters:

| | |
|---|---|
| F | (FILE) Identifies the file. |
| list of file attributes | One file attribute or a series of attributes and values. Common file attributes are listed below. |

List of Common File Attributes

| | |
|---|---|
| ACCESS_MODE = list of keyword value | KEY_POSITION = integer |
| CHARACTER_CONVERSION = boolean | KEY_TYPE = keyword value |
| DATA_PADDING = integer | MAXIMUM_BLOCK_LENGTH = integer |
| EMBEDDED_KEY = boolean | MAXIMUM_RECORD_LENGTH = integer |
| FILE_CONTENTS = keyword value | MESSAGE_CONTROL = list of keyword value |
| FILE_LIMIT = integer | OPEN_POSITION = keyword value |
| FILE_ORGANIZATION = keyword value | PADDING_CHARACTER = character |
| FILE_STRUCTURE = keyword value | PAGE_FORMAT = keyword value |
| INDEX_PADDING = integer | PAGE_LENGTH = integer |
| INTERNAL_CODE = keyword value | PAGE_WIDTH = integer |
| KEY_LENGTH = integer | RECORD_TYPE = keyword value |

For more information about these attributes, see the discussion of File Attributes in chapter 10, File Interface Introduction. See the SCL System Interface Usage manual for a detailed description of the SET_FILE_ATTRIBUTE command.

## CHANGE_FILE_ATTRIBUTE (CHAFA)

CHANGE_FILE_ATTRIBUTES changes the values of some file attributes for an existing file.

Simple examples:

```
/chafa  file=$user.myfile  file_contents=list
```

Changes file contents (FC) to LIST for permanent file MYFILE.

```
/chafa  file=current_file  file_structure=data
```

Changes file structure (FS) to DATA for local file CURRENT_FILE.

### CHANGE_FILE_ATTRIBUTE (CHAFA) Format

```
CHAFA F=file    list of file attributes
```

Parameters:

| | |
|---|---|
| F | (FILE) Specifies the permanent or local file whose attributes are to be changed. |
| list of file attributes | One or more file attributes with their new values. Some of the file attributes that can be changed are the following:<br><br>FILE_CONTENTS = keyword value<br>FILE_PROCESSOR = keyword value<br>FILE_STRUCTURE = keyword value |

For a complete list of file attributes whose values can be changed by the CHAFA command, see the SCL System Interface Usage manual.

## DISPLAY_FILE_ATTRIBUTE (DISFA)

DISFA displays the values of attributes for one or more files.

Simple examples:

The command:

```
/display_file_attributes  file=$user.sine_file ..
.. /display_options=(maximum_record_length  record_type  file_organization)
```

displays to $OUTPUT (see the File Connection Commands discussion later in this chapter) the maximum_record_length (MAXRL), record_type (RT), and file_organization (FO) of the permanent file SINE_FILE in your user catalog.

The command:

```
/display_file_attributes  file=new_temp ..
.. /output=my_result
```

sends information to the temporary file MY_RESULT about a default set of attributes for the temporary file NEW_TEMP.

The command:

```
/display_file_attributes  file=$user.old_test ..
.. /display_options=all ..
.. /output=$user.keep
```

sends to the permanent file KEEP information on the entire set of attributes for the permanent file OLD_TEST.

### DISPLAY_FILE_ATTRIBUTE (DISFA) Format

Below is a commonly used form of the DISPLAY_FILE_ATTRIBUTES command:

```
DISFA  F=file  DO=keyword  O=file
```

Parameters:

F (FILE) Specifies the file for which the selected attributes are to be displayed; can be a list of files. This parameter is required.

DO (DISPLAY_OPTION) Specifies the attribute or attributes that should be displayed. Any attribute specified in the SETFA command can be displayed using the DISFA command. Specifying ALL lists all the attributes. Omission causes a default set of attributes to be displayed.

O (OUTPUT) Specifies the file to which the display information is to be written. The default is standard file $OUTPUT, which is usually connected to file OUTPUT.)

## Interstate Connection Commands

The interstate connection allows you to execute commands through NOS while logged in on NOS/VE. The commands of the interstate connection are: CREIC (CREATE_INTERSTATE_CONNECTION), EXEIC (EXECUTE_INTERSTATE_COMMAND), and QUIT or DELIC (DELETE_INTERSTATE_CONNECTION). The following example uses the interstate connection. A FORTRAN 5 program produces a file OUTFILE, which is transferred to the NOS/VE state:

```
CREIC  <--------------------------- Create interstate connection.
EXEIC 'GET,PROGIN.'  <------------- Execute NOS GET command.
EXEIC 'DEFINE,OUTFILE.'  <--------- Execute NOS DEFINE command because only permanent files
                                    can be transferred.
EXEIC 'FILE,OUTFILE,BT=C,RT=Z.'  <-- Execute NOS FILE command that specifies a sequential CZ
                                    file.
EXEIC 'FTN5,I=PROGIN.'  <---------- Executes NOS FTN5 compiler.
EXEIC 'LGO.'  <-------------------- Executes FORTRAN program
DELIC  <--------------------------- Delete interstate connection.
GETF OUTFILE  <-------------------- NOS/VE command to transfer OUTFILE.
```

The interstate connection is primarily intended for use with the NOS/VE File Management Utility (FMU) to migrate files. These tasks require the GET, ATTACH, FILE, and REPLACE commands to be processed by NOS while NOS/VE handles the conversion processing.

For examples of using the interstate connection and FMU to migrate files, see the examples of using FMU in chapter 11, General Facilities for Migrating Files.

## CREATE_INTERSTATE_CONNECTION (CREIC)

The CREATE_INTERSTATE_CONNECTION command initializes the interstate connection. This command does not affect your local files on NOS/VE; they do not disappear. The format of the command is:

```
CREATE_INTERSTATE_CONNECTION  PARTNER_JOB_CARD='string'  STATUS=variable
```

PARTNER_JOB_CARD (PJC) is an optional parameter that specifies Job statement parameters that apply to NOS commands executed while the interstate connection is open. The syntax of the specified string must conform to NOS rules for Job statements. Refer to Volume 3 of the NOS 2 Reference Set for details on the format and values of Job statement parameters. Your site administrator might decide to have a default job card for every Interstate Communications job that is run. In this case, the PARTNER_JOB_CARD parameter given by the user is ignored.

STATUS is an optional parameter for a status variable; see the discussion of the STATUS parameter that appears later in this chapter.

When CREATE_INTERSTATE_CONNECTION is in effect, you can enter any NOS/VE command except commands, such as GET_FILE and REPLACE_FILE, that request a link to NOS, or another CREATE_INTERSTATE_CONNECTION command. You enter NOS/VE commands just as you would without CREATE_INTERSTATE_CONNECTION in effect. Only one interstate connection can be open at any given time.

The CREATE_INTERSTATE_CONNECTION command sets up a batch job in NOS to execute the NOS commands. If there are any accounting information (family name, user name, etc.) differences between NOS and NOS/VE, you might need to use the SET_LINK_ATTRIBUTE command. The SCL System Interface Usage manual discusses this command.

## EXECUTE_INTERSTATE_COMMAND (EXEIC)

The EXECUTE_INTERSTATE_COMMAND command precedes all NOS commands and can be specified only when CREATE_INTERSTATE_CONNECTION is in effect. The NOS commands, along with any error messages generated by those commands, are written to the NOS/VE job log. (You can display the job log with the DISPLAY_LOG command.) The format of the EXECUTE_INTERSTATE_COMMAND command is:

```
EXECUTE_INTERSTATE_COMMAND  COMMAND='command-string'  STATUS=variable
```

Parameters:

| | |
|---|---|
| Command-string | A string of NOS commands. The string must not exceed 240 characters and must be enclosed in apostrophes. You should generally specify only one NOS command in command-string so that you can determine which NOS command failed if an abnormal status is returned. However, to execute a NOS loader sequence, you must include all loader commands in the command-string of a single EXEIC. |
| STATUS | Status variable. |

Example:

```
/execute_interstate_command  command='GET,DATAFIL.'
```

## QUIT or DELETE_INTERSTATE_CONNECTION (DELIC)

Either the QUIT or DELETE_INTERSTATE_CONNECTION command terminates the interstate connection. Enter them as follows:

```
QUIT
```

```
DELIC
```

Optionally, the STATUS parameter can be used with DELETE_INTERSTATE_CONNECTION. See the discussion of the STATUS parameter that appears later in this chapter.

## File Connection Commands

NOS/VE automatically provides file connections for many files associated with your job. When you understand the file connections, you can control them. File connections can do the following things for you:

- Provide a list like the NOS dayfile of the NOS/VE commands that have been processed. This list is essential in debugging procedures. For example, the following CREATE_FILE_CONNECTION command writes a list to the file MYFILE of all NOS/VE commands processed after the command:

  ```
  /create_file_connection  standard_file=$echo ..
  .. /file=myfile
  ```

  This action is similar to entering the following NOS DAYFILE statement which creates a list of all NOS statements processed before the DAYFILE statement:

  ```
  DAYFILE,L=MYFILE.
  ```

- Let you specify the file or files that your program uses for input or output.

- Let you specify the error or list file.

To use file connections you need to understand the concept of standard files on NOS/VE.

### Standard Files

File connection works by connecting a standard file with a real file. Standard files are files that the system assumes for various functions. A standard file represents a file but has no space. To be used, a standard file must be connected to one or more real files (files with space).

For example, consider a FORTRAN program. The compiler default value for the LIST (L) parameter (which controls the listing of the source program) is the standard file $LIST. The real file to which the system writes the listing depends on how the program executes.

If the program executes interactively, the system automatically connects $LIST to $NULL (which causes information to disappear) so that you do not get excessive output written to your terminal. If a program executes in batch mode, the system automatically connects $LIST to file OUTPUT and writes the listing to OUTPUT.

All standard files have the dollar sign ($) for the first character of their names. The standard files that can be connected with a CREATE_FILE_CONNECTION (CREFC) command are:

| | |
|---|---|
| $COMMAND | Specifies the file from which SCL commands are currently being read. |
| COMMAND_OF_CALLER | Specifies the $COMMAND file from which an SCL procedure was called. (Once a procedure has been invoked, $COMMAND designates the file containing the procedure.) Chapter 6, Using Procedures, describes SCL procedures. |
| $ECHO | Receives command images as they are executed. |
| $ERRORS | Receives error information. |
| $INPUT | Identifies an input file. |
| $LIST | Receives output listings. |
| $OUTPUT | Receives output as designated by programs or commands. |
| $RESPONSE | Receives the responses (error or informative completion messages) from processed commands. |

Table 4-2 shows the real files to which the standard files are connected when you initially use NOS/VE.

Table 4-2. Initial Connection of Standard Files

| Standard File | Interactive Connection | Batch Connection |
|---|---|---|
| $COMMAND | COMMAND | COMMAND |
| $COMMAND_OF_CALLER | None | None |
| $ECHO | $NULL | $NULL |
| $ERRORS | OUTPUT | OUTPUT |
| $INPUT | INPUT | INPUT |
| $LIST | $NULL | OUTPUT |
| $OUTPUT | OUTPUT | OUTPUT |
| $RESPONSE | OUTPUT | $NULL |
| | Job Log | Job Log |

Notes:

$NULL is like a sink hole; information disappears in it.

Files INPUT and OUTPUT generally act the same as they do on NOS (the default files for input or output). $INPUT and $OUTPUT are standard files; INPUT and OUTPUT are real files. However, INPUT on NOS/VE is always an empty file in batch processing.

The job log contains log records of job processing. Use the job log by entering the SCL command DISPLAY_LOG (DISL). Procedure processing does not appear on the job log, just the call to the procedure appears.

## CREATE_FILE_CONNECTION (CREFC) Format

Below is a commonly used form of the CREATE_FILE_CONNECTION command:

```
CREATE_FILE_CONNECTION  STANDARD_FILE=standard file name  FILE=file
```

Parameters:

SF (STANDARD_FILE) can be one of the following (see preceding paragraphs for information on the standard files):

| | |
|---|---|
| $COMMAND | $INPUT |
| $COMMAND_OF_CALLER | $LIST |
| $ECHO | $OUTPUT |
| $ERRORS | $RESPONSE |

F (FILE) specifies the file name of the actual file to be connected.

The command to terminate the connection is DELETE_FILE_CONNECTION (DELFC).

## Example of the Dayfile Equivalent

To get a listing comparable to a dayfile listing, use the CREFC (CREATE_FILE_CONNECTION) command to connect standard file $ECHO to a file. $ECHO receives the images of commands as they are processed. Initially, this file is connected to $NULL and, therefore, contains no information.

Examples:

```
/create_file_connection  standard_file=$echo  file=dayfile1
```

Causes information to be written to file DAYFILE1 (this could be any file name).

```
/create_file_connection  standard_file=$echo  file=output
```

Causes information to be written to file OUTPUT.

Recommendation: Use the connection to $ECHO when debugging interactive procedures. This is the best way for you to determine where your procedure aborts.

The procedure below shows the CREFC command included in the command stream:

```
proc myproc  <---------- Procedure header for MYPROC.
crefc $echo dayfile1  <-- Creates the file connection.
...  <------------------ Other commands.
delfc $echo dayfile1  <-- Deletes the connection.
procend myproc  <-------- Terminates the procedure.
```

To list the contents of DAYFILE1 after the procedure executes, enter:

```
/copy_file  input=dayfile1
```

## Example of Designating Multiple Output Files

To get listings on multiple output files, use the CREFC (CREATE_FILE_CONNECTION) command to connect standard file $OUTPUT to several files. Code your program to designate $OUTPUT as the output file. In the command stream before executing the program, connect $OUTPUT to other files. When the program executes, NOS/VE writes to each file connected to $OUTPUT.

A sample job that uses $OUTPUT to have three files written is shown in figure 4-1. The FORTRAN program specifies writing to $OUTPUT. But $OUTPUT is connected to the following file: local file OUTPUT, local file REPORT, and permanent file MSGLOG. The commands to create and delete the file connections appear in the command stream.

```
"NOTE: OUTPUT connected."  <-------- $OUTPUT/OUTPUT automatically connected.
crefc $output report  <------------- Connection to local file REPORT.
crefc $output $user.msglog.$eoi  <-- Connection to end-of-information position of permanent
                                     file MSGLOG.

Program  <-------------------------- FORTRAN program executes.

      OPEN (2,FILE='$OUTPUT')  <---- Designates output on file $OUTPUT.
      ..
      WRITE (2, ('1X,  ...  <------- Writes to $OUTPUT.

      CLOSE (2)  <------------------ Program closes the file.

delfc $output report  <------------- Deletes file connection.
delfc $output $user.msglog  <------- Deletes file connection.
copf report  <---------------------- Copies file contents to terminal.
copf $user.msglog  <---------------- Copies file contents to terminal.
```

Figure 4-1. Sample FORTRAN Program Using $OUTPUT

## DELETE_FILE_CONNECTION (DELFC)

DELETE_FILE_CONNECTION deletes the connection between one of the standard files and the real file. Remembering to delete file connections can be important and are easy to forget if a procedure creates a file connection but aborts before deleting the connection.

If unexpected things happen when processing on NOS/VE, consider a forgotten file connection, and delete it.

For example, assume that your executing program gets data through standard file $INPUT (which is used on NOS/VE instead of file INPUT on NOS). $INPUT is empty in batch execution; it must be connected to a real file with data. Assuming the real file is INDATA, the commands are:

```
/create_file_connection  standard_file=$input  file=indata  <-- Creates the file connecion.
/lgo  <------------------------------------------------------ Executes the program.
/delete_file_connection  standard_file=$input  file=indata  <-- Deletes the file connection.
                                                               This is important because
                                                               other commands using $INPUT
                                                               will use data in INDATA until
                                                               the connection is deleted.
```

### DELETE_FILE_CONNECTION (DELFC) Format

Below is a commonly used form of the DELETE_FILE_CONNECTION command:

```
DELETE_FILE_CONNECTION  STANDARD_FILE=standard file name  FILE=file
```

Parameters:

SF (STANDARD_FILE) can be one of the following:

| | |
|---|---|
| $COMMAND | $INPUT |
| $COMMAND_OF_CALLER | $LIST |
| $ECHO | $OUTPUT |
| $ERRORS | $RESPONSE |

F (FILE) specifies the file name of the actual file to be disconnected.

## DISPLAY_FILE_CONNECTION (DISFC)

DISPLAY_FILE_CONNECTION displays the names of the file currently connected to the standard files.

For example:

```
/display_file_connection
```

Displays all standard files together with the files connected to them.

```
/display_file_connection  standard_file=$errors
```

Displays the files connected to standard file $ERRORS.

### DISPLAY_FILE_CONNECTION (DISFC) Format

Below is a commonly used form of the DISPLAY_FILE_CONNECTION command:

```
DISPLAY_FILE_CONNECTION  STANDARD_FILE=standard file name  OUTPUT=file
```

Parameters:

SF (STANDARD_FILE or FILES) Specifies any or all of the following standard files: $COMMAND, $COMMAND_OF_CALLER, $ECHO, $ERRORS, $INPUT, $LIST, $OUTPUT, $RESPONSE. The default is the keyword ALL, which causes the display of all standard file connections.

O (OUTPUT) Specifies the file to which information is written. Default is $OUTPUT, which is usually connected to file OUTPUT.

# Miscellaneous Common Commands

The miscellaneous common NOS/VE commands discussed in this manual perform important functions in debugging and structuring jobs. The commands are:

**COLLECT_TEXT**

**DISPLAY_VALUE**

**PUT_LINE**

## COLLECT_TEXT (COLT)

COLLECT_TEXT provides NOS/VE users with a way to specify the NOS equivalent of an end-of-record (EOR) in your NOS/VE command stream. Actually, COLT causes subsequent text to be written to the specified file. The text could be a program, data, commands--literally any text. COLT works this way:

```
/collect_text  file=myfile  <-- Identifies the file to which text is written.
ct? text  <-------------------- Text for file MYFILE.
ct? more text 1
ct? more text n
ct? **  <---------------------- Two asterisks terminate COLT operation.
```

COLLECT_TEXT is a very useful command for building jobs and procedures on NOS/VE.

Figure 4-2 shows a sample NOS procedure that you can create in NOS/VE by using COLLECT_TEXT. The procedure executes a FORTRAN program that reads names from unit INPUT and then prints them. The procecure executes in interactive mode.

```
.PROC,LSTNME.  <-------------- Identifies the NOS procedure.
FTN5.  <---------------------- Compiles program; NOS finds program file after EOR.
LGO.  <----------------------- Executes program; NOS finds input file after EOR.
-EOR-  <---------------------- End-of-record indicator.
        PROGRAM FTNPG  <------ FORTRAN source program.
        CHARACTER *10 NAME
        DO 10, I=1,100
        READ (*,'(A)') NAME
    10  PRINT *, NAME
        END
-EOR-  <---------------------- End-of-record indicator.
CHARLES  <-------------------- Data.
DIANA
 ...  <----------------------- Represents more data.
-EOR-  <---------------------- End-of-record indicator.
```

Figure 4-2. Sample NOS Procedure

There are several ways to use the COLLECT_TEXT command to set up the sample NOS procedure to execute on NOS/VE. The resulting NOS/VE procedure is shown in figure 4-3. The procedure executes interactively and creates a file connection to $INPUT.

```
proc list_name  <---------- Identifies the procedure.
colt filea  <-------------- Identifies the file to which COLT writes the source program.
        PROGRAM FTNPG  <-- FORTRAN source program.
        CHARACTER *10 NAME
        DO 10, I=1,100
        READ (*,'(A)') NAME
    10  PRINT *, NAME
        END
**  <---------------------- Terminates COLT operation.
colt infile  <------------- Identifies file to which COLT writes data.
Charles  <----------------- Data.
Diana
 ...  <-------------------- Represents more data.
**  <---------------------- Terminates COLT operation.
fortran i=filea  <--------- File FILEA is input for the FORTRAN compiler.
crefc $input infile  <----- To make READ* work (uses $INPUT), create a file connection
                            between standard file $INPUT and file INFILE.  See the File
                            Connection discussion earlier in this chapter for more
                            information.
lgo   <-------------------- Executes FORTRAN binary on file LGO.
delfc $input infile  <----- Deletes the file connection.  This is important because other
                            commands could use INFILE for input until you delete the
                            connection.
procend list_name  <------- Terminates the procedure.
```

Note: If the procedure resides by itself on a file, you can execute it by specifying the file name.

Figure 4-3. Sample NOS/VE Procedure Uses COLLECT_TEXT

### COLLECT_TEXT (COLT) Format

Below is a commonly used form of the COLLECT_FILE command:

```
COLLECT_FILE  O=file
```

Parameters:

O (OUTPUT) Identifies the file to which the collected text is written. You can designate a local or permanent file.

## DISPLAY_VALUE (DISV)

DISPLAY_VALUE displays a value or list of values. DISPLAY_VALUE is comparable to the DISPLAY and NOTE commands on NOS. This is a good command to use in debugging interactive jobs. It can also display information obtained from SCL functions.

Examples:

```
/display_value  value=$catalog
```

Displays the working catalog.

```
/display_value  value=$date
```

Display the current date.

```
/display_value  value=' job step 2 failed; processing continues.'
```

Writes the message to file OUTPUT.

```
/display_value  value=(' sum=',5+2)  output=outfile
```

Writes SUM = 7 to file OUTFILE.

Used in batch mode, this command causes a new page to be printed. To avoid the paging, you can use either the PUT_LINE command or the SET_FILE_ATTRIBUTES command to set the PAGE_FORMAT attribute of the output file to NON_BURSTABLE. For example, the following command avoids paging the OUTFILE used in the last DISPLAY_VALUE example:

```
/set_file_attributes  file=outfile  page_format=non_burstable
```

### DISPLAY_VALUE (DISV) Format

Below is a commonly used form of the DISPLAY_VALUE command:

```
DISPLAY_VALUE  V=value-list  O=file
```

Parameters:

V (VALUE) Specifies the value or list of values to be displayed. Enclose a list in parentheses. Enclose literals in apostrophes. When more than one value is specified, each is displayed on a separate line.

O (OUTPUT) Specifies the file to which the information is displayed. The default is standard file $OUTPUT. (See the File Connection commands discussion earlier in this chapter for more information.)

If this command creates the output file, it sets the PAGE_FORMAT file attribute to BURSTABLE. See the PAGE_FORMAT attribute discussion in chapter 10, File Interface Introduction, for more information.

## PUT_LINES (PUTL)

PUT_LINES writes a line or lines to a file. PUT_LINES is comparable to the NOTE command on NOS. This is a good command to use in documenting batch jobs when you need to write messages about processing performed. Since the first character of each string is interpreted as carriage control, you can control print spacing, page ejects, etc.

Example:

```
/put_lines lines=' JOB STEP 2 FAILED; PROCESSING CONTINUES.'
```

Writes the message to file OUTPUT.

### PUT_LINES (PUTL) Format

Below is a commonly used form of the PUT_LINES command:

```
PUT_LINES  L=lines  O=file
```

Parameters:

L (LINES) specifies the line or lines as strings to be written to a file. Enclose strings in apostrophes. Enclose a list of strings in parentheses. The first character of each line is used for carriage control for that line.

O (OUTPUT) Specifies the file to which the information is written. The default is standard file $OUTPUT. (See the File Connection commands discussion earlier in this chapter for more information.)

# NOS/VE STATUS Parameter

All NOS/VE commands have an optional STATUS parameter. When you specify this parameter, a previously declared variable of kind STATUS (all SCL variables must be declared as being of a kind) is specified to contain status information.

By checking the value of the status variable, you can alter processing, depending on status conditions. The status variable contains status information in the following fields:

| | |
|---|---|
| NORMAL | A boolean value of FALSE or TRUE, depending on the command processing incorrectly or correctly, respectively. |
| CONDITION | A unique code indicating the processor detecting the error and the specific error detected. |
| TEXT | A string (up to 256 characters). |

The status variable provides a powerful mechanism in handling errors. However, describing its use is not within the scope of this manual; it is an optional parameter.

See the SCL Language Definition Usage manual for details on using this feature.

## CONTEXT Differences

Both NOS and NOS/VE support the CYBER Online Text System (CONTEXT). CONTEXT is conceptually identical on both systems. It enables you to read, create, and modify online manuals. However, individual commands and directives differ. For a complete description of NOS/VE CONTEXT, refer to the CONTEXT Usage manual. The following is a summary of the differences between the NOS and NOS/VE versions of CONTEXT. The complete description of these differences is in the CONTEXT Usage manual.

### Reading an Online Manual

The NOS and NOS/VE commands to read an online manual are as follows:

```
NOS:      EXPLAIN or REVIEW,M=bound-file
                            L=listing-file-name  <--Use with REVIEW command only
                            *CHILD=YES
                            subject?

NOS/VE:   EXPLAIN
               SUBJECT='subject'
               MANUAL=bound-file
               LIST=excerpt-file
               STATUS=status-variable
```

On both systems, all parameters are optional. If you omit the bound-file value, you are taken to a default manual that contains general information about using the NOS or NOS/VE system.

Both systems allow you to go directly to a particular subject within a manual by specifying the subject on the EXPLAIN command. If you omit the subject, you are taken to the first screen (main menu) of the manual.

The LIST parameter on the NOS/VE EXPLAIN command specifies a file to receive the screen information sent when you use Copy and SetLog. NOS provides this capability through the REVIEW command and the L parameter.

The *CHILD=YES parameter on the NOS EXPLAIN command is used for manuals that contain embedded procedures.

The HELP command also provides you access to an online manual, usually the SCL Quick Reference manual.

The commands for navigating through an online manual are similar on both systems:

| | |
|---|---|
| + or RETURN | Displays the next screen. |
| - | Displays the preceding screen. |
| Up | Displays the menu from which you made your last selection. |
| Top or First | Displays the first screen (main menu) of the manual. |
| Find | Searches for screen describing a topic. |
| Back | Displays screen from which you used Find. |
| Index | Displays a portion of the manual's index. |
| Help | The first time you use Help, displays information on how to use the keys that perform online manual reading operations. If you immediately use Help again, displays the current HELP screen. |
| Quit | Exits the online manual and returns to system command mode. |

NOS/VE CONTEXT provides some commands and capabilities that are not provided by NOS CONTEXT, and vice versa. These are as follows:

Under NOS/VE, you can read online manuals in the full-screen mode. This mode makes it easier for you to navigate through the manuals, because you can do the navigation by pressing keys instead of by entering commands. NOS does not have this feature.

Under NOS/VE, you can enter SCL commands (including another EXPLAIN command) while reading an online manual. This is not permitted under NOS.

On NOS/VE, you can make a menu selection to branch to another online manual. Using Revert returns you to the original manual. NOS does not provide this capability, although it can be simulated through embedded procedures.

NOS provides the REVIEW and PRINT commands, which enable a reader to copy selected screens, together with comments on those screens, to a file. On NOS/VE, using Copy and SetLog provides a similar capability by copying a portion of an online manual to a file.

In NOS/VE CONTEXT, using Refrsh redisplays the current screen. NOS CONTEXT does not provide this capability.

## Creating an Online Manual

The commands and directives used in the creation of an online manual differ on NOS and NOS/VE.

### Directives

The directives used to structure an online manual have different names on NOS/VE. The NOS/VE directive names follow standard SCL naming conventions. Each directive name has a long form and an abbreviation. In addition, NOS/VE provides several new directives. The following table lists the NOS CONTEXT directives and the equivalent NOS/VE directives. NOS/VE abbreviations are shown in parentheses.

| NOS Directives | NOS/VE Directives | Description |
|---|---|---|
| \S | \CREATE_SCREEN (\CRES) | Defines and names a screen. |
| \U | \DEFINE_UP (\DEFU) | Designates "upward" screen. |
| \P | \DEFINE_PRIOR (\DEFP) | Designates "prior" screen. |
| \N | \DEFINE_NEXT (\DEFN) | Designates "next" screen. |
| \M | \DEFINE_MENU (\DEFM) | Defines menu selections. |
| \I | \DEFINE_INDEX_TOPIC (\DEFIT) | Defines index topic. |
| \C | \"..." | Comment. |
| \H | \DEFINE_LOCAL_HELP_SCREEN (\DEFLHS) | Defines screens reached via HELP command. |

The following directives are provided by NOS/VE but not by NOS.

| Directive | Description |
|---|---|
| \DEFINE_MANUAL_LINK (\DEFML) | Establishes a link to another online manual. |
| \DEFINE_MANUAL_NAME (\DEFMN) | Defines the name of an online manual. |
| \DEFINE_DIRECTIVE_CHAR (\DEFDC) | Changes the directive character from \ to a specified character. |
| \DEFINE_ALTERNATE_COMMAND (\DEFAC) | Changes the name of a command. |
| \DEFINE_SCL_PROCEDURE (\DEFSP) | Signals the beginning of an embedded procedure. |
| \DEFINE_SCL_PROCEDURE_END (\DEFSPE) | Signals the end of an embedded procedure. |
| \EXECUTE_SCL_PROCEDURE (\EXESP) | Begins execution of an embedded procedure. |
| \DELETE_SCL_PROCEDURE (\DELSP) | Deletes an embedded procedure. |

Differences in the rules for writing directives are as follows:

- On NOS/VE, a screen name consists of 1 through 31 characters. On NOS, the screen name must not exceed 10 characters.
- On NOS/VE, the priority level that you assign to an index topic is 0 through 15, and can occur 16 times. Thus, an index topic can occur a total of 256 times. On NOS, the priority levels are 0 through 9, and can occur 10 times.
- On NOS/VE, an index topic can be up to 256 characters long. On NOS, index topics are limited to 50 characters.

### Binding an Online Manual

The NOS and NOS/VE commands for creating an online manual are as follows:

```
NOS:     BINDER,I=source-file,O=bound-file,L=list-file,OUTPUT=error-file.

NOS/VE:  CREATE_MANUAL
             INPUT=source-file
             OUTPUT=bound-file
             ERROR_LIST=error-file
             LIST=cross-reference-file
             LIST_OP=R or NONE
             STATUS=status-variable
```

CREATE_MANUAL can be abbreviated CREM. CREATE_MANUAL does not provide a formatted listing of the online manual like the one produced by the L parameter on the BINDER command.

The LIST and LIST_OP parameters on the CREATE_MANUAL command provide a cross-reference listing of the manual. This is similar to the listing produced by the OMREF utility under NOS. The formats of the NOS and NOS/VE cross-reference listings differ slightly.

# Debugging on NOS and NOS/VE

Debug is a NOS/VE interactive debugging facility that is similar to NOS CYBER Interactive Debug (CID). This discussion compares the most commonly used features of the two debugging facilities.

## Basic Concepts

Both Debug and CID provide powerful interactive debugging capabilities. Debug and CID are conceptually similar; they both allow you to control and monitor the execution of a program. Specifically, you can use the debugging facilities to:

Suspend program execution at specified locations or on the occurrence of specified conditions.

Display the values of program variables while execution is suspended.

Change the values of program variables while execution is suspended.

Automatically execute sequences of commands on the occurrence of specified conditions.

Resume execution after execution has been suspended.

Both facilities allow you to reference program locations symbolically; that is, by variable name or line number.

Debug provides symbolic support for FORTRAN, C, COBOL, BASIC, Pascal, and CYBIL. Unlike CID, Debug does not provide language-dependent commands for each language. The set of Debug commands is the same for all supported languages.

## Full Screen Debugging

Debug can operate in full screen mode. It uses cursor movement and certain keys to operate on data displayed on the terminal screen.

Each key used to perform a full screen debugging operation is associated with a command. The commands include the following:

- Debug commands, such as SET_BREAK
- Screen commands, such as Fwd
- Help
- Quit

Pressing one of these keys performs the action assigned to that key.

The screen for full screen mode debugging is divided as follows:

| | | |
|---|---|---|
| 1. | Home line | The line on which you enter Debug commands and SCL commands |
| 2. | Response line | The line where Debug displays short responses and advisory messages |
| 3. | Source window | The area for displaying the program you are debugging |
| 4. | Output window | The area for showing the output created by your program as well as the output you requested from Debug |
| 5. | Menu rows | The area displaying the assignments to the keys performing full screen debugging operations. |

## Command Format

Although Debug and CID perform similar operations, the commands differ between the two facilities. The most noticeable difference is in the command formats. Debug commands conform to SCL command syntax. Consistent with this syntax, each Debug command has one of the following forms:

```
verb_object              For example, SET_BREAK

verb_modifier_object     For example, DISPLAY_PROGRAM_VALUE
```

In addition, each command contains parameters that select options associated with the command. Like SCL commands, all Debug commands and parameters also have an abbreviated form.

Following are examples of Debug commands. Each command is shown twice; the first time in its full form and the second time using abbreviations. The command:

```
set_break  break=aa  line=26
setb  b=aa  l=26
```

sets a break named AA at line 26 of the program unit.

The command:

```
display_program_value  name=svar  module=sub4
```

displays the value of the variable SVAR in program unit SUB4.

For more information about SCL command syntax, refer to chapter 2, Conventions for Commands, Names, and Parameters.

## Home Program

When debugging a program with multiple modules, you can specify the module to which address references in commands apply. (In FORTRAN, a module is a program unit; in COBOL, a module is identified by the program-name specified in the PROGRAM-ID paragraph of the IDENTIFICATION division.)

In CID, you use the notation P.module-name to specify a module. In Debug, you specify the module name in a command parameter. For example, the following CID and Debug commands are equivalent; both display the value of the variable A in module MYSUB.

```
CID:      DISPLAY,P.MYSUB_A.

Debug:    display_program_value  name=a  module=mysub
```

For any Debug command that references an address (line number or variable name), the MODULE parameter specifies the module that contains the address.

If you omit the MODULE parameter from a Debug command, the command assumes the default module. The default module in Debug is the same as the home program in CID; the initial default module is the module in which execution is currently suspended. You can change the default module by entering a CHANGE_DEFAULT command (equivalent to the SET,HOME command of CID).

For example:

```
change_default  module=mysub  (abbreviated chad  m=mysub)
```

changes the default module to MYSUB. Thereafter, all commands that do not contain a MODULE parameter reference locations in module MYSUB.

The new default module remains as specified in the CHANGE_DEFAULT command until you either enter another CHANGE_DEFAULT command or end the debug session. The command to change the default module back to the current module is:

```
change_default  module=$current
```

The following example illustrates two ways of referencing a location in a different module.

```
program z
a=1.0
call sub
end  <------------ If execution is suspended here,

subroutine sub
a=2.0  <---------- and you want to display the value of A as defined here,
return
end
```

then, enter either of the following:

```
display_program_value  name=a  module=sub  <-- Specifies module SUB
```

or

```
change_default  module=sub  <----------------- Changes default module to SUB
display_program_value  name=a
```

## Steps for Using Debug and CID

To use DEBUG or CID, proceed as follows:

1. Compile for use with Debug. (For FORTRAN, use the following commands. For other languages, refer to the appropriate language manual or to the Debug Usage manual.)

   NOS: `FTN5, I=program, DB=ID`

   NOS/VE: `fortran  input=program  optimization_level=debug  debug_aids=dt`

2. Turn on debug mode.

   NOS: `DEBUG,ON.`

   NOS/VE: `set_program_attributes debug_mode=on`

3. Begin the debug session.

   NOS: `LGO.`

   NOS/VE: `lgo`

4. Set traps and breakpoints.

   NOS:
   ```
   SET,BREAKPOINT,L.n.  <--- Set breakpoint at line n.
   SET,BREAKPOINT,S.n.  <--- Set breakpoint at statement n.
   SET,TRAP,LINE,*.  <------ Set trap at each line.
   ```

   NOS/VE:
   ```
   SET_BREAK L=n  <--------- Set break at line n.
   SET_BREAK SL=n  <-------- Set break at statement label n.
   SET_STEP_MODE MODE=ON <-- Set step mode for each executable line.
   ```

5. Begin program execution.

```
NOS:      GO.

NOS/VE:   run
```

6. Display or change program values. (PRINT is a FORTRAN CID command. For other languages, refer to the CID reference manual.)

```
NOS:      DISPLAY,variable.  <-------------------------- Display the value of the variable.
          PRINT,list.  (FORTRAN only)
          ENTER,value,variable.  <---------------------- Assign new value to the variable.

NOS/VE:   DISPLAY_PROGRAM_VALUE  NAME=variable  <------- Display value of program variable.
          CHANGE_PROGRAM_VALUE  N=variable  V=value  <-- Assign new value to program
                                                         variable.
```

7. End the debug session.

```
NOS:      QUIT.

NOS/VE:   quit
```

## Preparing for a Debug Session

Both Debug and CID require a special compilation to produce the symbol and line number tables required for symbolic debugging. In NOS/VE compiler languages, generation of these tables is controlled by the DEBUG_AIDS parameter on the compiler call command, which is similar to the DB parameter on NOS compiler calls. Additionally, you should also specify OPTIMIZATION_LEVEL=DEBUG on the compiler call command.

For example:

```
NOS:      FTN5,I=MYPROG,DB=ID.

NOS/VE:   fortran  input=myprog  debug_aids=dt  optimization_level=debug
```

These commands compile the source program on file MYPROG and produce tables for symbolic debugging. IF DEBUG_AIDS=DT is not specified on the FORTRAN command, symbolic debugging is not possible.

After compiling your program with debugging tables generated, the next step is to turn on debug mode. On NOS/VE you can turn on debug mode by specifying the following command:

```
set_program_attributes debug_mode=on  (abbreviated setpa dm=on)
```

This command is equivalent to the DEBUG,ON control statement under NOS; all subsequent program executions will take place under Debug control until you either end the terminal session or turn off debug mode. The command to turn off debug mode is:

```
set_program_attributes debug_mode=off  (abbreviated setpa dm=off)
```

Recall that on NOS, specifying DEBUG,ON before compilation obviates the need to specify DB=ID on the compiler call. On NOS/VE, SET_PROGRAM_ATTRIBUTES has no effect on compilation.

## Beginning and Ending a Debug Session

Beginning a debug session is the same under NOS and NOS/VE; simply type the name of the binary object file while debug mode is on. The system responds with a prompt indicating that you can now enter Debug commands. For example:

```
NOS:     LGO.

         CYBER Interactive Debug
         ?

NOS/VE:  lgo

         debug
         DB/
```

On NOS/VE, the Debug prompt for user input is DB/.

When Debug issues the DB/ prompt, you can enter any Debug or SCL command. Typically, at the beginning of the session, you will enter commands that will cause program execution to suspend at some point during the session. Suspending execution is discussed in the next topic.

The Debug command to begin (or resume) program execution is:

```
run
```

The RUN command is equivalent to the CID GO or EXECUTE command, except that it does not allow you to specify an address where execution is to begin. (RUN always begins execution at the point of suspension.)

In both CID and Debug, the command to end a debug session and return to system command mode is:

```
QUIT
```

## Suspending Program Execution

Under both NOS and NOS/VE, the first step in a debug session is generally to provide for gaining control during the session. Both CID and Debug provide commands for specifying conditions or locations at which execution will suspend so that you can enter other commands.

Under both systems, you can begin execution and allow the program to execute until an error occurs. If an execution error occurs, execution automatically suspends and you can enter Debug commands. As explained in a later topic, both systems also allow you to specify a sequence of commands that is automatically executed when an execution error occurs.

Debug provides the equivalent of traps and breakpoints. In Debug, the mechanism for suspending execution is called a break. You can set a break at a statement label or line number (similar to a CID breakpoint) or you can set a break for a specified event (similar to a CID trap), such as a call to a subprogram.

The Debug command to establish a break is SET_BREAK (abbreviated SETB). Parameters on the SET_BREAK command specify the type and location of the break. In addition, each break is assigned a name. You can assign the name yourself through an optional parameter or you can allow Debug to assign a default name. The name has the same function as the trap or breakpoint number in CID; that is, you can use it to reference the break in subsequent commands.

Following are typical SET_BREAK commands (abbreviations are shown in parentheses):

```
set_break line=10  (setb l=10)
```

Sets a break at line 10 of the current module. This is equivalent to setting a CID breakpoint at line 10 of the program.

```
set_break event=call procedure=mysub  (setb e=c p=mysub)
```

Sets a CALL break in the current module. A CALL break is similar to an RJ trap; it suspends execution when a call to the specified procedure (subprogram) is encountered.

```
set_break break=b3 statement_label=5 module=sub  (setb b=b3 sl=5 m=sub)
```

Sets a break at the statement labeled 5 in module SUB. The parameter BREAK=B3 assigns the name B3 to the break.

Another useful method of suspending execution is to turn on step mode. Step mode is used to suspend execution at the beginning of a specified unit of code, thus allowing you to step through the execution of a program.

For example, the following command is similar to the CID LINE trap; it causes execution to suspend at the beginning of each executable line of code.

```
SET_STEP_MODE MODE=ON UNIT=LINE MODULE=name
```

where name is the name of the module in which step mode is set. The MODULE parameter can be omitted, in which case the current (default) module is assumed.

SET_STEP_MODE can be abbreviated SETSM.

You can also set step mode to suspend execution at the beginning of each procedure or program unit of a program. The command is:

```
set_step_mode mode=on unit=procedure
```

Use the following command to turn off step mode:

```
SET_STEP_MODE MODE=OFF MODULE=name
```

For COBOL programs, you can also use step mode to perform the same function as the PROCEDURE trap of CID. The Debug commands:

```
set_step_mode mode=on unit=cobol_paragraph  (abbreviated setsm mode=on u=cp)

set_step_mode mode=on unit=cobol_section  (abbreviated setsm mode=on u=cs)
```

Set step mode so that execution suspends at the beginning of each paragraph in the PROCEDURE division or each section, respectively.

## Displaying Program Values

The Debug command for displaying the values of data items within a program while execution is suspended is DISPLAY_PROGRAM_VALUE (abbreviated DISPV). This command is similar to the CID DISPLAY command; it specifies a variable, array, or array element to be displayed. Parameters on the DISPLAY_PROGRAM_VALUE command specify the data item to be printed and the program containing the data item.

Following are some examples:

```
display_program_value name=x
```

Displays the value of the variable X in the default module.

```
display_program_value name=beta module=mysub
```

Displays the value of the variable BETA in module MYSUB.

```
display_program_value name=arr(5)
```

Displays the value of element 5 of array ARR in the default module.

If you specify an array name on a DISPLAY_PROGRAM_VALUE command, the entire array is displayed.

The following form of DISPLAY_PROGRAM_VALUE is similar to the LIST,VALUES command of CID; it displays all values in the specified module (or in the current module if the MODULE= parameter is omitted).

```
DISPLAY_PROGRAM_VALUE NAME=$ALL MODULE=module-name
```

DISPLAY_PROGRAM_VALUE is the only Debug command for displaying values. There are no language dependent commands such as the PRINT (FORTRAN, BASIC) and MAT PRINT (BASIC) commands of CID.

## Changing Program Values

The Debug CHANGE_PROGRAM_VALUE command (abbreviated CHAPV) is similar to the CID ENTER command; it assigns a new value to a program variable, array, or array element. Parameters on the CHANGE_PROGRAM_VALUE command specify the item to be changed and the value to be assigned to the item. Some examples are as follows:

```
change_program_value name=x value=3.14  (abbreviated chapv n=x v=3.14)
```

Assigns the value 3.14 to the variable X in the current program unit.

```
change_program_value name=alpha(1) value=.00247 module=proga
```

Assigns the value .00247 to the first element of array ALPHA in program unit PROGA.

CHANGE_PROGRAM_VALUE is the only Debug command for changing values of variables. Debug does not provide language dependent commands such as the assignment statements of CID.

## Other Debug Features

The following is a comparison of some additional features of Debug and CID.

### Automatic Execution of Commands

The SET_BREAK command of Debug allows you to specify a sequence of commands to be executed automatically when the break occurs, in a manner similar to the trap and breakpoint bodies of CID. The optional COMMAND parameter (abbreviated C) on the SET_BREAK command specifies the commands to be executed. You can specify a string of one or more Debug commands.

For example, the following SET_BREAK command specifies a single DISPLAY_PROGRAM_VALUE command to be executed when the break is reached during execution. Note that the DISPLAY_PROGRAM_VALUE command is enclosed in apostrophes.

```
set_break break=brk line=4 command='display_program_value name=x'
```

The following SET_BREAK command specifies four DISPLAY_PROGRAM_VALUE commands to be executed when the break is reached. Note that the command string is continued on a second line and that commands are separated by a semicolon. For more information about command continuation, refer to chapter 2, Conventions for Commands, Names, and Parameters.

```
set_break break=b004 line=10 command='display_program_value name=a; ..
display_program_value name=b; display_program_value name=c; display_program_value name=d'
```

Debug provides a method of automatically executing a sequence of commands on the occurrence of an execution error. This capability is similar to a CID ABORT trap with a body. To use automatic command execution, do the following:

1. Create a file of Debug commands. You can create this file using a text editor such as the NOS/VE Full Screen Editor. (The file can also contain SCL commands.)

2. Enter the command:

```
SET_PROGRAM_ATTRIBUTES DEBUG_MODE=OFF ABORT_FILE=file
```

   where file is the file of Debug and SCL commands.

3. Begin program execution.

The program will execute until an error occurs. The commands in the ABORT_FILE file will then be automatically executed. Note that DEBUG_MODE must be set to OFF in order to use the ABORT_FILE feature.

Debug has capabilities similar to CID groups. You can execute SCL command procedures or use the INCLUDE_FILE command in a debug session. Thus, you can create sequences of debug and SCL commands, execute the commands at will during a debug session, and save the commands for use in a later session.

This capability is extremely powerful, because you can use the full capabilities of the SCL command language in Debug command sequences. For example, you can use the SCL block structuring statements to provide for repeated or conditional execution of command sequences.

Command procedures, block structuring statements, and the INCLUDE_FILE command are described in the SCL Language Definition Usage manual. Refer to chapter 6, Using Procedures, for an introduction to SCL procedures.

The following command sequence uses an IF/IFEND block to specify conditional execution of commands:

```
if $current_line = 15 then
  display_program_value name=a
  display_program_value name=b
else
  display_program_value name=x
  display_program_value name=y
ifend
```

The IF statement tests the $CURRENT_LINE function. If the current line is 15, the values of variables A and B are displayed. Otherwise, the values of variables X and Y are displayed.

If these commands were placed in a file named AAA, they could be executed during a debug session by the following command:

```
include_file file=aaa
```

The following example combines the command string capability of the SET_BREAK command with the INCLUDE_FILE feature.

```
set_break line=10 command='display_program_value name=x; display_debugging_environment; ..
include_file file=aaa; run'
```

When program execution reaches line 10, execution suspends and the following occurs:

The value of the variable X is displayed.

The debugging environment is displayed.

The commands in file AAA are executed.

Program execution resumes.

Specifying command strings on a SET_BREAK command is especially useful for debugging DO loops.

### Displaying Debug Status Information

Debug provides commands for displaying information about the status of a debug session. The DISPLAY_BREAK command is similar to the LIST,TRAP and LIST,BREAKPOINT commands of CID for displaying information about traps and breakpoints currently active in the debug session.

Examples of the DISPLAY_BREAK command are as follows:

```
display_break break=(b1,b2,b3)  (abbreviated disb b=(b1,b2,b3))
```

Displays the names, types, and locations of breaks B1, B2, and B3. (B1, B2, and B3 are the names assigned in the SET_BREAK command.)

```
display_break break=all  (abbreviated disb b=all)
```

Displays the names, types, and locations of all breaks currently active in the debug session.

The DISPLAY_DEBUGGING_ENVIRONMENT command displays a paragraph of information about the current debug session, similar to that displayed by the CID LIST,STATUS command. Optional parameters on the DISPLAY_DEBUGGING_ENVIRONMENT command select types of information to be displayed. Entering

```
display_debugging_environment  (abbreviated disde)
```

with no parameters, produces a default display. The most useful information given by this display is the location (line number and program unit) where execution is currently suspended.

Debug provides several functions, similar in purpose to the CID debug variables, that return information about the debug session. The more commonly used Debug functions are:

| | |
|---|---|
| $CURRENT_LINE | Returns the line where execution is suspended. |
| $CURRENT_MODULE | Returns the name of the program unit where execution is suspended. |
| $CURRENT_PROCEDURE | Returns the name of the procedure where execution is suspended (for programs other than FORTRAN or COBOL.) |

You can display the values returned by these functions by entering a DISPLAY_VALUE command in the debug session. For example:

```
display_value value=$current_line
```

Displays the line where execution is currently suspended.

The Debug functions can also be used in expressions in command sequences. For example:

```
if $current_line = 6 then
   display_program_value name=x
   display_program_value name=y
   display_program_value name=z
ifend
```

If the current line is 6, then the values of variables X, Y, and Z are displayed.

## Displaying a Subprogram Traceback List

The Debug DISPLAY_CALL command is equivalent to the TRACEBACK command of CID. The DISPLAY_CALL command (abbreviated DISC) lists the sequence of called subprograms beginning with the current subprogram and proceeding back to the main program.

Optional parameters on the DISPLAY_CALL command specify the subprogram where the traceback is to begin and whether the traceback is to include system as well as user subprograms.

The following examples show two forms of the DISPLAY_CALL command:

```
display_call display_options=all_calls (disc do=ac)
```

Displays a traceback list beginning with the current subprogram. The list includes both system and user subprograms.

```
display_call display_options=user_calls  (disc do=uc)
```

Displays a traceback list beginning with the current subprogram. The list includes only user subprograms.

## Removing Breaks

The Debug DELETE_BREAK command (abbreviated DELB) removes breaks from the current debug session. This command serves the same purpose as the CID CLEAR,BREAK and CLEAR,TRAP commands. You can remove selected breaks or all breaks.

For example:

```
delete_break break=(b1, b4, b6)
```

Deletes breaks B1, B4, and B6 from the current debug session. (B1, B4, and B6 are the names assigned by the SET_BREAK command).

```
delete_break all
```

Deletes all breaks from the debug session.

## CID and Debug Commands and Features

The following list summarizes the CID and Debug commands and features.

| CID Command or Feature | Equivalent Debug Command or Feature |
|---|---|
| ABORT trap | Debug automatically gets control when an execution error occurs. |
| assignment statement | No language-dependent commands. Use CHANGE_PROGRAM_VALUE. |
| Bodies | Specify a command string with the COMMAND parameter on the SET_BREAK command. |
| Breakpoint at statement label or line number. | Break at statement label or line number. |
| CLEAR,BREAKPOINT | DELETE_BREAK |
| CLEAR,GROUP | Not needed. |
| CLEAR,TRAP | DELETE_BREAK |

| CID Command or Feature | Equivalent Debug Command or Feature |
|---|---|
| COLLECT mode | Not needed. Command sequences can be created by a text editor or COLLECT_TEXT. |
| DISPLAY | DISPLAY_PROGRAM_VALUE |
| ENTER | CHANGE_PROGRAM_VALUE |
| EXECUTE | RUN |
| GO | RUN |
| Groups | Use INCLUDE_FILE or command procedures to execute predefined sequences of commands. |
| HELP | No equivalent. Read online manual while in debug session: EXPLAIN MANUAL=DEBUG. |
| INTERPRET mode | No equivalent. |
| LET | No language-dependent commands. Use CHANGE_PROGRAM_VALUE. |
| LINE trap | Use SET_STEP_MODE to suspend execution ateach executable line. |
| LIST,BREAKPOINT | DISPLAY_BREAK |
| LIST,GROUP | Not needed. |
| LIST,MAP | No equivalent. |
| LIST,STATUS | DISPLAY_DEBUGGING_ENVIRONMENT |
| LIST,TRAP | DISPLAY_BREAK |
| LIST,VALUES | DISPLAY_PROGRAM_VALUE NAME=$ALL |
| MAT PRINT | No language-dependent commands. Use DISPLAY_PROGRAM_VALUE. |
| MESSAGE | Use DISPLAY_VALUE 'string'. |
| MOVE | No equivalent. |
| OVERLAY trap | No equivalent; overlays not supported on NOS/VE. |
| PRINT | No language-dependent commands. Use DISPLAY_PROGRAM_VALUE. |
| PROCEDURE trap | Use SET_STEP_MODE MODE=ON UNIT=PROCEDURE or, for COBOL: SET_STEP_MODE MODE=ON UNIT=COBOL_PARAGRAPH |
| QUIT | QUIT |
| RJ trap | SET_BREAK BREAK=name EVENT=CALL |
| SAVE | No equivalent. |
| Sequence commands (SKIPIF, LABEL, JUMP, READ) | Use SCL block structuring statements. |
| SET,AUXILIARY | No equivalent. |
| SET,BREAKPOINT | Use SET_BREAK to set breaks at specified line numbers or statement labels. |
| SET,HOME | Use CHANGE_DEFAULT to change default program. |

| CID Command or Feature | Equivalent Debug Command or Feature |
|---|---|
| SET,INTERPRET | No equivalent. |
| SET,OUTPUT | No equivalent. |
| SET,TRAP | Use SET_BREAK to set breaks at specified events. |
| STEP | SET_STEP_MODE |
| SUSPEND | No equivalent. |
| TRACEBACK | DISPLAY_CALLS |
| variables | Use Debug functions to get information about debug session. |
| VETO mode | No equivalent |

### Where To Go for More Debug Information

Two manuals describe the NOS/VE Debug facility:

Debug Usage manual (printed)

Contains detailed descriptions and formats of all Debug commands and functions

Debug Quick Reference (online, type EXPLAIN MANUAL=DEBUG)

Contains brief descriptions and formats of all Debug commands and functions

In addition, the printed manual for each language supported by Debug contains a short example of a Debug session and a reference to the Debug Usage manual. The online manual for each such language links from its main menu to the online Debug Quick Reference manual. These languages are:

FORTRAN
BASIC
C
COBOL
CYBIL
Pascal

## Sort/Merge Differences

NOS/VE provides a Sort/Merge utility that is compatible with NOS Sort/Merge Version 5. In most cases, NOS programs that use Sort/Merge 5 can be executed on NOS/VE with little or no modification.

### Summary of Major Differences

The following paragraphs describe the major differences between NOS Sort/Merge 5 and NOS/VE Sort/Merge.

#### Byte Size

A NOS byte contains 6 bits. A NOS/VE byte contains 8 bits.

### Character Data

NOS character data is internally represented in 6-bit display code. NOS/VE character data is internally represented in 8-bit ASCII code. The ASCII character set is shown in appendix C.

### Character Sets

NOS supports the 63- and 64-character sets. NOS/VE supports only the 256-character ASCII character set.

### Collating Sequences

NOS/VE supports the following predefined collating sequences (the associated NOS/VE names are shown in parentheses):

```
ASCII6 (OSV$ASCII6_FOLDED)
COBOL6 (OSV$COBOL6_FOLDED)
DISPLAY (OSV$DISPLAY64_FOLDED)
EBCDIC6 (OSV$EBCDIC6_FOLDED)
EBCDIC (OSV$EBCDIC)
ASCII (default; no associated NOS/VE name)
```

Although the above names are the same as the NOS names, the actual collating sequences may differ.

NOS/VE does not support the INTBCD collating sequence.

The NOS/VE collating sequences are given in appendix C.

User-defined collating sequences under NOS have 64 positions. User-defined collating sequences under NOS/VE have 256 positions. When converting a NOS Sort or Merge program to NOS/VE, you can use the SM5SEQR procedure (or SEQR command parameter) to fill the extra positions.

### Diagnostic Messages

The error numbers and message text of NOS/VE Sort/Merge error messages differ from those of NOS Sort/Merge 5. The NOS/VE Sort/Merge error messages are listed in the Diagnostic Messages for NOS/VE manual. (They are not listed in the manuals that describe Sort/Merge.)

Sort/Merge 5 writes messages to the dayfile and to the file specified by the E parameter (or SM5E procedure call). NOS/VE Sort/Merge writes messages to the files specified by the LIST parameter (or SM5LIST call) and by the E parameter (or SM5E call).

### Equal Keys

When Sort/Merge encounters two records with equal key values, it calls the owncode 5 procedure if you have specified one. The procedure has the choice of retaining, deleting, or altering either or both records. It cannot alter key fields. A fatal error occurs if you use this procedure with summing or with the OMIT_DUPLICATES option.

### Estimated Number of Records

Under NOS/VE, you can specify a value for the ENR parameter or the SM5ENR call, but the value is not used. ENR and SM5ENR are provided only for compatibility with Sort/Merge 5.

### Exception File Processing

Exception file processing is a new feature provided by NOS/VE. The ERF parameter or SM5ERF procedure call allows you to specify a file to which invalid records are written during a sort or merge operations.

### FASTIO Processing

Using the FASTIO parameter in the SM5FAST procedure causes Sort/Merge 5 to read and write directly rather than through the access method. NOS/VE provides the SM5FAST procedure for compatibility with NOS. NOS/VE does not support the FASTIO parameter. If you use the SM5FAST procedure in NOS/VE, a warning message is issued.

### File Attributes

Unlike NOS, under NOS/VE, not all of the default file attributes are valid for a sort or merge operation.

Under NOS/VE, the default value for the MINIMUM_RECORD_LENGTH attribute could cause a fatal error if no key field was specified for the sort or merge.

Sort and merge require a value for the maximum record length, even for record manager record types that do not require this specification. The MAXIMUM_RECORD_LENGTH parameter of the SET_FILE_ATTRIBUTES command must specify the maximum record length value for files which do not have F type records if the default of 256 is not large enough.

### File Positioning

Sort/Merge 5 rewinds files before use, depending on the type of file or unless a FILE statement parameter specifies otherwise. NOS/VE Sort/Merge does not rewind files. The open position of a NOS/VE file is determined by the value of its OPEN_POSITION attribute. This attribute is established by a SET_FILE_ATTRIBUTES command.

### Interactive Prompting

NOS/VE Sort/Merge does not provide the capability of entering Sort or Merge command parameters in response to interactive prompts.

### Listing File

When accessed through procedure calls, Sort/Merge 5 writes listable output to file OUTPUT. NOS/VE Sort/Merge allows you to specify an output listing file through the SM5LIST procedure call. The default listing file is $LIST.

### MERGE Command

The formats of the NOS/VE MERGE command and parameters follow different conventions from those of NOS MERGE. The MERGE command and parameters are described later in this chapter.

### Owncode Procedures

NOS/VE Sort/Merge does not provide an OWNF parameter to specify a file containing owncode routines. Under NOS/VE, any owncode procedures specified in a SORT or MERGE command must be accessible from an object library in the current object library list.

On NOS/VE, letters in owncode names must be uppercase.

### Procedure Calls

Both NOS/VE Sort/Merge and NOS Sort/Merge 5 support FORTRAN procedure calls as well as a system command interface. In addition, NOS/VE Sort/Merge is callable through the CYBIL language. The FORTRAN procedure call differences are described later in this chapter. The CYBIL procedure calls are described in the CYBIL Keyed File and Sort/Merge Interfaces Usage manual.

### Sign Overpunch

Sort/Merge 5 defines 20 sign overpunches. NOS/VE Sort/Merge defines 34 sign overpunches.

### Sort 4 Support

NOS Sort 5 supports the Sort 4 FORTRAN interface; NOS/VE Sort does not support the Sort 4 FORTRAN interface.

### SORT Command

The formats of the NOS/VE SORT command and parameters follow different conventions from those of the SORT5 command and parameters. The SORT command and parameters are described later in this chapter.

### STATUS Parameter

In Sort/Merge 5, a status code is returned in the specified CCL variable. In NOS/VE Sort/Merge, this is the standard SCL status parameter, in which status information is returned. The SCL STATUS parameter is described in chapter 4, Common NOS/VE Commands. NOS/VE does not support the ST alias for STATUS.

### Zero Comparison

In Sort/Merge 5, negative zero is ordered before positive zero. In NOS/VE Sort/Merge, positive and negative zero are ordered equally.

## FORTRAN-Sort/Merge Procedure Call Differences

Both NOS Sort/Merge 5 and NOS/VE Sort/Merge can be called through system commands and from FORTRAN programs through procedure calls. In addition, NOS/VE Sort/Merge can be called from CYBIL, a new high-level language provided by NOS/VE. This topic describes FORTRAN procedure call differences. The next major topic describes system command (control statement) differences. Refer to the CYBIL Keyed File and Sort/Merge Interfaces Usage manual for descriptions of the CYBIL procedure calls.

The Sort/Merge 5 and NOS/VE Sort/Merge procedure calls are compatibile; they have the same names and parameters. However, in some cases parameter values differ, and NOS/VE provides several new calls.

The differences between the NOS and NOS/VE Sort/Merge procedure calls follow. Only calls for which differences exist are described. Refer to the FORTRAN Language Definition Usage manual for detailed descriptions of all of the Sort/Merge procedure calls.

### SM5E

If this call is omitted, the NOS default error file is the output listing; the NOS/VE default is file $ERRORS.

### SM5EL

Under NOS, error levels are specified as T or 1, W or 2, R or 3, and C or 4. NOS/VE allows T or I, W, F, and C, but not 1, 2, 3, or 4. The default is I. Specifying NONE requests that no error messages be listed.

### SM5ENR

NOS/VE provides this call for compatibility with NOS, but it has no effect.

### SM5FAST

NOS/VE provides this call for compatibility with NOS. If you call SM5FAST, NOS/VE does nothing except issue a warning message.

### SM5KEY

On NOS/VE, SM5KEY supports the following collating sequence names: ASCII6, COBOL6, DISPLAY, EBCDIC6, ASCII, and EBCDIC. Even though the first four names are the same as those supported on NOS, the actual collating sequences might differ. Note that SM5KEY does not support the INTBCD collating sequence.

On NOS/VE, a collating sequence specified in a call to SM5KEY must be predefined or defined by a call to SM5DUCT, SM5LCT, or SM5SEQx.

### SM5NODA

NOS/VE provides this call for campatibility with NOS. If you call SM5NODA, there is no effect.

### SM5OWNn

For Sort/Merge 5, owncode routines are specified by the address of the owncode routine name that was declared in an EXTERNAL statement. For NOS/VE Sort, an owncode procedure is specified by the entry point name. Letters in the name must be uppercase and the entry point must be loadable by PMP$LOAD.

### SM5ST

The NOS/VE SM5ST procedure specifies a status variable in which the completion status of the command or procedure is returned.

### SM5SUM

The NOS/VE Sort/Merge summing capability combines all records having equal key fields into a single record. The record is chosen to be the longest record of all the records being summed. Sort/Merge deletes the rest of the records during summing. If several of the records being combined have the maximum length, then, which record is retained and which records are deleted is undefined. The NOS/VE SM5SUM procedure specifies a sum field in the retained record. The sum field is the sum of the values of the corresponding fields from the combined records. You may not call SM5SUM with SM5OWN5, SM5OMIT, or SM5RETA in the same sort or merge.

### New FORTRAN Sort/Merge Procedure Calls

NOS/VE provides the following new FORTRAN Sort/Merge procedure calls:

| | |
|---|---|
| SM5CC | Controls the format of some of the parameter names. |
| SM5DUCT | Specifies the name of a user-defined collating table to be used by SM5KEY. |
| SM5ERF | Specifies a file to which all invalid records encountered during the sort or merge operation are written. |
| SM5FMA | Allows you to pass memory areas to be used as if they were FROM files. |

SM5LCT Loads a collation table for use in a SM5KEY call.

SM5LIST Specifies the name of a file to receive the Sort/Merge output listing.

SM5LO Selects the type of information to be written to the output listing.

SM5OMIT Controls omitting all but one of the records which have equal key values. You cannot call SM5OMIT with SM5OWN5 in the same sort or merge. The SM5OMIT, SM5RETA, and SM5SUM calls are mutually exclusive.

SM5TMA Allows you to pass memory areas to be used as if they were a TO file.

SM5VER Requests checking of order of input records for a merge operation.

SM5ZLR Controls the use of zero length records.

## SORT and MERGE Command Difference

Both NOS Sort/Merge 5 and NOS/VE Sort/Merge can be called by a system control statement (command). The NOS/VE SORT and MERGE commands are equivalent to the NOS SORT5 and MERGE control statements. However, the formats of the commands and their associated parameters differ between the two systems.

The formats of the NOS SORT5 and MERGE commands are as follows:

```
SORT5.keyword1=value1...keywordn=valuen

MERGE.keyword1=value1...keywordn=valuen
```

The NOS/VE SORT and MERGE commands conform to SCL conventions. The formats of these commands are as follows:

```
SORT keyword1=value1...keywordn=valuen

MERGE keyword1=value1...keywordn=valuen
```

On both NOS and NOS/VE, commas or blanks are used to separate parameters.

Both the NOS and NOS/VE Sort/Merge commands can be continued over multiple lines. The maximum length of the NOS SORT 5 and MERGE commands is 240 characters; NOS/VE SORT and MERGE commands can be any length. Command continuation rules are described in chapter 2, Conventions for Commands, Names, and Parameters.

Both NOS and NOS/VE provide the option of specifying Sort/Merge parameters in a directive file.

Parameters on the NOS/VE SORT and MERGE commands conform to SCL conventions; each parameter has a full form and, in most cases, an abbreviation. For ease of use, the equivalent NOS parameter name is usually supported as well.

Following is an example of equivalent NOS and NOS/VE commands to call Sort/Merge.

```
NOS:     SORT5.FROM=SRTIN TO=SRTOUT KEY=1..10

NOS/VE:  sort from=srtin to=srtout key=1..10
```

These commands read input records from local file SRTIN, write output records to file SRTOUT, and define the first 10 bytes as the key field.

Both NOS and NOS/VE allow parameters to be specified positionally (however, the position is not always the same). For example:

NOS: SORT5. SRTIN SRTOUT 1..10

NOS/VE: SORT SRTIN SRTOUT 1..10

The following is a list of NOS and NOS/VE Sort/Merge parameters. The NOS/VE parameter abbreviations are shown in parentheses.

| NOS Parameter | NOS/VE Parameter | Description |
|---|---|---|
| FROM | FROM (F) | Specifies file containing input records to be sorted or merged. |
| TO | TO (T) | Specifies output file to receive sorted or merged records. |
| KEY | KEY (K) | Defines the record key field to be used in the sort or merge operation. Also selects a collating sequence. NOS default sequence is ASCII6; NOS/VE default is ASCII. |
| DIR | DIRECTIVES_FILE (DIRECTIVES, DIR, DF) | Specifies a file containing Sort/Merge directives. |
| L | LIST (L) | Specifies a file to receive the output listing. NOS default is OUTPUT; NOS/VE default is $LIST. |
| LO | LIST_OPTIONS (LO) | Specifies output listing options. |
| E | ERROR (E) | Specifies a file to receive error messages. NOS default is output listing file; NOS/VE default is $ERRORS. |
| EL | ERROR_LEVEL (EL) | Specifies the level of errors to be written to error file. |
| DIALOG (DIA) | No equivalent | NOS/VE does not support interactive prompting. |
| ENR | ESTIMATED_NUMBER_RECORDS (ENR) | Estimated number of input records; not used on NOS/VE. |
| OWNF | No equivalent | Specifies file containing owncode routines. On NOS/VE, owncode routines must be in an object library. |
| OWNFL (OFL) | OWNCODE_FIXED_LENGTH (OWNFL, OFL) | Specifies length of fixed length records input from owncode routine. |
| OWNMRL (OMRL) | OWNCODE_MAXIMUM_RECORD_LENGTH (OWNMRL, OMRL) | Specifies maximum length of owncode records. |
| OWNn | OWNCODE_PROCEDURE_n (OWNn, OPn) | Specifies owncode procedure. |
| RETAIN (RET) | RETAIN_ORIGINAL_ORDER (RETAIN, RET, ROO) | Directs Sort/Merge to output records with equal keys in same order as input. This parameter, OMIT_DUPLICATES, and SUM are mutually exclusive. |
| SEQx | COLLATING_SEQUENCE_x ($CS_x$) | Defines a collating sequence. |

| NOS Parameter | NOS/VE Parameter | Description |
|---|---|---|
| STATUS (ST) | STATUS | Specifies variable to receive error status code. Format of information returned differs between NOS and NOS/VE. |
| SUM | SUM (S) | Specifies record field to be summed. This parameter, OMIT_DUPLICATES, and RETAIN_ORIGINAL_ORDER are mutually exclusive. You cannot use OWNCODE_PROCEDURE_5 with SUM. |
| VERIFY (VER) | VERIFY_MERGE_INPUT_ORDER (VERIFY, VER, VMIO) | Requests verification of order of input records to be merged. |
| FASTIO | No equivalent | Direct input and output not supported on NOS/VE. |

The following new parameters are supported by the NOS/VE SORT and MERGE commands.

C170_COMPATIBLE (CC)

When you set this parameter to ON, YES, or TRUE, owncode routine names are processed as upper case letters. When you set the parameter to OFF, NO, or FALSE, the names are processed in the form you entered them. The default is OFF.

EXCEPTION_RECORDS_FILE (ERF)

Specifies a file to which all invalid records encountered during a sort or merge operation are written.

LOAD_COLLATING_TABLE (LCT)

Loads a collating table.

OMIT_DUPLICATES (OD)

Controls omitting all but one of the records with equal key values. The record that remains is chosen to be the longest record of all the records with equal key values. If several of the records with equal key values have the maximum length, then, which record remains and which records are deleted is undefined. This parameter, RETAIN_ORIGINAL_ORDER, and SUM are mutually exclusive. You cannot use OWNCODE_PROCEDURE_5 with OMIT_DUPLICATES.

RESULT_ARRAY (RESA, RA)

Specifies an array in which Sort/Merge returns processing statistics.

ZERO_LENGTH_RECORDS (ZLR)

Controls the use of zero length records.

## Where To Go for More Information About Sort/Merge

The Sort/Merge procedure calls, and usage information about calling Sort/Merge from languages, is presented in the language usage manuals. Calling Sort/Merge with system commands is discussed in the SCL Advanced File Management Usage manual. Manuals with Sort/Merge information are also available in online form on NOS/VE. The online manuals can be accessed through EXPLAIN.

| Manual Title | Online Description |
|---|---|
| COBOL Usage Manual | EXPLAIN M=COBOL S='SORT' |
| SCL Advanced File Management Usage Manual | EXPLAIN M=AFM S='SORT' |
| FORTRAN Quick Reference Manual | EXPLAIN M=FORTRAN S='SORT' |

# Job Structure 5

# Job Structure 5

This chapter describes how you can use various SCL statements and commands to structure jobs under NOS/VE. In previous chapters, you were introduced to some SCL commands for performing various useful operations under NOS/VE. Many of your jobs under NOS/VE will probably consist of simple sequences of these commands. In addition to these commands, SCL provides statements that enable you to organize a job into blocks of commands such that each block performs a single logical function. You can specify blocks to be executed repeatedly until certain conditions are satisfied, or you can specify blocks that are executed only when certain errors occur.

There are some important differences in the structures of NOS and NOS/VE jobs. SCL has a more natural, language-like structure that is more familiar to FORTRAN and COBOL programmers. Under NOS, you have limited use of such elements as variables and operators, which you can combine into expressions and various types of statements. Under NOS/VE, you have much more flexibility in the use of these elements, which you can use to construct jobs that resemble structured programs.

The SCL block structuring statements include statements such as IF/IFEND, BLOCK/BLOCKEND, and WHILE/WHILEND.

A job block begins at the beginning of a job and ends at the end of the job. Within a job block you can define other blocks. The following example shows a job consisting of two blocks within a job block:

```
LOGIN
      ------------------------------------------------
BLOCK       --------------------------------
  CREATE_VARIABLE NAME=SVAR KIND=STATUS      }
  ATTACH_FILE FILE=$USER.TEST STATUS=SVAR    }  block
BLOCKEND    --------------------------------

IF SVAR.NORMAL THEN ------------------------
    FORTRAN INPUT=TEST LIST=FLIST          }
ELSE                                       }  block
    DISPLAY_VALUE VALUE='ERROR IN JOB'     }
IFEND               ------------------------

LOGOUT      ------------------------------------------------
```

job block

## SCL Language Elements

An SCL job is composed of elements that can be combined into expressions and statements. The elements are: constants, variables, expressions, assignment statements, and functions.

### Constants

A constant represents a value that does not change during a job. NOS provides numeric and literal constants. NOS/VE provides three types of constants: integer, string, and boolean. The following table compares NOS and NOS/VE constants:

| NOS | NOS/VE |
|---|---|
| Numeric (maximum length = 10 digits). | Integer (maximum size = 281474976710655). |
| Literal (character string enclosed in dollar signs; maximum length = 10 characters or 40 characters, depending on how the string is to be used). | String (maximum length = 256 characters). |
| - | Boolean (has the logical value TRUE or FALSE). |

## Variables

A variable is a symbol that contains a value that can be changed during a job. NOS provides three registers, designated as R1, R2, and R3, that function as variables. You can assign values to these registers and subsequently change, display, or test those values. NOS also provides special symbols, such as the error flags EF and EFG, into which you can store certain values.

Under NOS/VE, you can define any number of variables and assign your own name to them. You can define a variable either implicitly (by referencing the variable in a statement) or explicitly (through the CREATE_VARIABLE command).

Every SCL variable has a property called the kind. The variable kinds are string, integer, boolean, and status. The kind of a variable is established when the variable is defined.

For example, the following assignment statement defines a variable named VAR and stores the sum of two existing variables into it:

```
VAR = FIRST_VAL + SECOND_VAL
```

In this case, the kind of the variable is determined by the kind of the value stored into it.

You can also define variables by using the following command:

```
CREATE_VARIABLE VARIABLE=name KIND=kind
```

where name is a name you assign to the variable, and kind is STRING, INTEGER, BOOLEAN, or STATUS.

SCL also provides commands that allow you to display or test the values of variables.

The following examples compare the use of NOS registers and NOS/VE variables:

```
NOS:      SET,R1 = R1 + R2.
          IF,R1 .LT. 10, LAB.
              DISPLAY, R1.
              SET,R1 = 0.
          ENDIF,LAB.

NOS/VE:   A = A + B
          IF A < 10 THEN
              DISV A
              A = 0
          IFEND
```

In both of these examples, a calculation is performed and the result is tested. If the condition is satisfied, two commands are executed; otherwise, the commands are skipped. Under NOS/VE, assignment statements rather then SET statements are used to assign values to variables.

## Expressions

An expression is composed of variables or constants separated by operators. Both systems allow you to write arithmetic, relational, and logical expressions.

SCL provides the same arithmetic operators as NOS: +, -, *, /, and **. SCL also provides a concatenation operator that joins two character values. The NOS/VE concatenation operator is indicated by the symbol //. Some examples of SCL expressions are:

```
J + 1

AVAR*BVAR**EXP

FIRST_STRING//SECOND_STRING//'END'
```

The last example joins two string variables and a string constant into a single string.

Under both NOS and NOS/VE, you can form relational and logical expressions. The relational and logical operators under NOS/VE are:

```
logical: OR, XOR, AND, NOT

relational:  =   equal to
             <   less than
             >   greater than
            <=   less than or equal to
            >=   greater than or equal to
            <>   not equal to
```

Logical and relational expressions are generally used in SCL block structured statements to control the flow of execution of a job. For example:

```
IF I < J THEN
  commands
IFEND
```

If the value of I is less than the value of J, the commands between IF and IFEND are executed. Otherwise, those commands are skipped. (The IF statement and other structured statements are discussed later in this chapter.)

## Assignment Statements

Under NOS, you use the SET command to assign values to symbolic names. Under NOS/VE, you use assignment statements to assign values to variables. The general form of an assignment statement is:

```
variable=expression
```

Some examples are:

| | | |
|---|---|---|
| NOS: | SET,R1=R1 + R2. | Sets R1 to the sum of R1 and R2 |
| NOS/VE: | XVAL = A + BB | Sets XVAL to the sum of A and BB. |
| | NEW_STRING = CHARS//'abc' | Concatenates the value of CHARS and the string abc and places the result in NEW_STRING. |

## Functions

Both NOS and NOS/VE provide functions that return values to your job. NOS provides the functions FILE, NUM, and DT. In addition, NOS provides a set of symbolic names in which the system places information about the status of the job. You can display the contents of these names or use them in expressions.

The SCL functions provide similar information under NOS/VE. You reference an SCL function by specifying the function name and arguments, if any, in a statement or command. The system then substitutes the value of the function for the function name. For example, the following commands display the current date:

```
NOS:     DISPLAY,DATE.

NOS/VE:  display_value   value=$date
```

The NOS example displays the contents of the DATE symbol, and the NOS/VE example displays the value of the $DATE function.

The following table shows the correspondence between some of the NOS functions and symbolic names and the SCL functions.

| NOS | NOS/VE |
|---|---|
| FILE (tests a file attribute) | $FILE (returns a file attribute) |
| NUM (determines whether a string is numeric) | Use the $VALUE_KIND function within a procedure to determine the kind of a value. |
| DT (determines device type for a file) | Use the $FILE function to retrieve the value of the DEVICE_CLASS attribute. |
| EF, EFG error flags | Use STATUS parameter on SCL commands. |
| DATE | $DATE |
| TIME | $TIME |
| MONTH | Use $SUBSTR($DATE(MONTH)1, 3). |
| HID | $PROCESSOR (identifies hardware in use). |
| FAMILY | $JOB(FAMILY_NAME) |

Other functions provided by SCL include the following:

| | |
|---|---|
| $ACCESS_MODE | Returns the access permissions of a file. |
| $CATALOG | Returns the current working catalog. |
| $CHAR | Converts an integer value to a character value. |
| $CLOCK | Returns the value of the microsecond clock. |
| $FNAME | Converts a string to a file name. |
| $INTEGER | Converts a string or boolean value to integer. |
| $STRLEN | Returns the length of a character string. |
| $STRREP | Converts any kind of value to a string. |

SCL also provides functions, such as the $VALUE_KIND function, that are valid only in procedures. These functions are discussed in chapter 6, Using Procedures. Refer to the SCL Language Definition Usage manual for complete descriptions of all the SCL functions.

The following example compares the NOS and NOS/VE functions for referencing file attributes:

```
NOS:      IF, FILE(MYFILE, .NOT. BOI), LAB.
              NOS statements ...
          ENDIF, LAB.
```

The FILE function returns a true value if the specified file has the specified attribute. Otherwise, the function returns a false value. In this example, if file MYFILE is not positioned at BOI, then the NOS statements between the IF and the ENDIF are executed.

```
NOS/VE:   IF $FILE(MYFILE, OP) <> '$BOI' THEN
              commands
          IFEND
```

The $FILE function returns the value of the specified attribute. In this example, the $FILE function tests the OP (OPEN_POSITION) attribute of file MYFILE. If the function value is not $BOI, then the statements between IF and IFEND are executed.

## Comparison of NOS and NOS/VE Elements

The following table compares the NOS and NOS/VE elements:

| NOS | NOS/VE |
|---|---|
| Numeric and literal constants | Integer, string, and boolean constants |
| Registers R1, R2, and R3 | User-defined variables |
| Arithmetic operators + - * / ** | Arithmetic operators + - * / ** |
| - | Concatenation operator // |
| Logical operators .OR. .AND. .XOR. .NOT. | Logical operators OR XOR AND NOT |
| Relational operators .EQ. .NE. .LT. .GT. .LE. .GE. | Relational operators = <> < > <= >= |
| SET command | Assignment statement |
| FILE function | $FILE function |
| DT function | DEVICE_CLASS parameter of $FILE function |
| NUM function | $VALUE_KIND function in procedures |
| DATE symbolic name | $DATE function |
| TIME symbolic name | $TIME function. |
| MONTH symbolic name | Use $SUBSTR($DATE(MONTH)1, 3) |
| EF, EFG symbolic names | Use STATUS parameter on SCL commands |
| FAMILY symbolic name | Use $JOB(FAMILY_NAME) |

## Controlling Job Flow

Under NOS/VE, a job consists of all processing that occurs from the time you log in to the time you log out. The SCL statements you enter control this processing. SCL provides statements for controlling the flow of a job by specifying repeated execution of commands, conditional execution of commands, and error condition processing. These statements are especially useful in SCL procedures, discussed in chapter 6.

### Block Structure

The block structuring capability of SCL differs significantly from the structure of NOS jobs. Under NOS, you use the IF statement to perform a relational test and to alter the flow of execution based on the result of the test. You use the WHILE statement to specify repeated execution of a sequence of statements.

NOS/VE extends the capability of the WHILE statement so that you can specify post-conditional repetition of blocks of statements.

Both NOS and NOS/VE enable you to specify unlimited repetition and preconditional repetition of blocks of statements, as well as conditional execution of blocks of statements without the need for branching.

Both systems enable you to test for error conditions during job processing and to specify processing to be performed if errors occur.

### Repeated Execution of Commands

Under both NOS and NOS/VE, you can specify loops, or repeated execution of sequences of commands. Under NOS, you use the WHILE statement to specify loops. Under NOS/VE, you can specify loops through the following structured statements:

FOR/FOREND

Specifies controlled repetition of commands.

WHILE/WHILEND

Specifies preconditional repetition of commands. (Similar to the NOS WHILE statement.)

REPEAT/UNTIL

Specifies post-conditional repetition of commands.

LOOP/LOOPEND

Specifies unlimited repetition of commands. (Similar to the NOS WHILE statement.)

#### FOR Statement

The FOR statement specifies controlled repetition of commands. That is, the commands are repeated a specified number of times. The FOR statement has the form

```
FOR variable = initial TO final DO
    commands
FOREND
```

At the beginning of this block, the variable is set to the specified initial value. On each pass through the block of commands, the variable is incremented by one until it attains or exceeds the final value. Control then passes to the command following the FOREND statement.

For example, the following NOS statements copy a file ten times:

```
SET,R2=1.
WHILE,R2 .LE. 10, LOOP1.
    REWIND,MASTER.
    COPY,MASTER,COPY10.
    SET,R2=R2+1.
ENDW,LOOP1.
```

NOS/VE does the same copying by using a FOR statement:

```
for i=1 to 10 do
    copy_file  input=master  output=copy10.$eoi
forend
```

The variable I is a user-defined variable used to control the number of times the loop is repeated. On each pass through the loop, I is incremented by 1 until it attains a value of 10. The loop is executed one more time (10 times total), and processing continues with the command following FOREND.

### WHILE Statement

SCL provides a WHILE statement that is similar to the NOS WHILE statement. A WHILE statement has the following general form:

```
WHILE expression DO
    commands
WHILEND
```

At the beginning of each pass through the block, the expression is tested. While the expression is true, the commands are repeated. Repetition continues until the expression attains a false value. Then control passes to the command following WHILEND.

Following are NOS and NOS/VE examples of a loop that is repeated five times:

```
NOS:     SET,R1=0.
         SET,R2=5.
         WHILE,R1 .LT. R2, FINISH.
           SET,R1=R1+1.
           DISPLAY,R1.
           .
           .
           .
         ENDW,FINISH.

NOS/VE:  I=0
         J=5
         WHILE I < J DO
           I=I+1
           DISPLAY_VALUE I
           .
           .
           .
         WHILEND
```

In the NOS example, register R1 is used for incrementing and testing within the loop. In the NOS/VE example, the variable I is used for incrementing and testing.

### REPEAT Statement

The REPEAT statement specifies post-conditional execution of a block of commands. That is, the specified condition is tested at the end of the block. The simplest form of the REPEAT statement is:

```
REPEAT
  commands
UNTIL expression
```

For example:

```
repeat
    accept_line  variable=line  input=input
    display_value  value=line
until line='DONE'
```

In this example, a line of input is read from the terminal (file INPUT) and displayed. Then the input line is tested. If its value is 'DONE', processing continues with the command following the UNTIL statement. If not, the block is repeated.

### LOOP Statement

A third way of looping under NOS/VE is through the LOOP statement:

```
LOOP
   commands
LOOPEND
```

These are comparable to the following NOS statements:

```
WHILE, TRUE, LOOP.
    NOS statements ...
    IF, condition. SKIP(LOOPEXIT).
ENDW,LOOP.
ENDIF,LOOPEXIT.
```

These statements specify repeated execution of a sequence of statements with no terminating conditions specified. In order to exit from this loop, you must specify an EXIT statement within the loop. For example:

```
loop
    accept_line  variable=str  input=input
    display_value  value=str  output=output2
    exit  when str='DONE'
loopend
```

This loop reads a line from the terminal and displays the line. If the line has the value 'DONE', control passes to the statement following LOOPEND. Otherwise, the loop is repeated.

## Conditional Execution of Commands

Both NOS and NOS/VE provide statements that enable you to conditionally execute sequences of commands. On NOS, you used the IF and ENDIF statements (or .IF and .ENDIF). On NOS/VE, you use IF and IFEND commands to construct blocks having the following form:

```
IF (expression) THEN
   commands
IFEND
```

If the expression is true, the commands between the IF and IFEND statements are executed. If the expression is false, those commands are skipped and processing continues with the commands following IFEND.

The following NOS and NOS/VE examples test to determine whether a file is assigned to a job. If so, a FORTRAN program is compiled and the output listing is written to the assigned file. If not, the compilation is skipped.

```
NOS:      IF,FILE(MYFILE,AS),AFFIRM.
               FTN5,I=SOURCE,L=MYFILE.
          ENDIF,AFFIRM.

NOS/VE:   if $file(myfile,assigned) then
               fortran  input=source  list=myfile
          ifend
```

In the NOS/VE example, the $FILE function tests the ASSIGNED attribute of file MYFILE.

## Error Condition Processing

Under both NOS and NOS/VE, you can specify action to be taken when error conditions occur during job processing.

Under NOS, you use the EXIT, NOEXIT, and ONEXIT statements to alter the flow of execution on the occurrence of errors. You also test the EF and EFG symbols and perform a desired sequence of operations based on the outcome of the test.

Under NOS/VE, there are two ways of handling error conditions. The first way is to test the status parameter provided by SCL commands. The second way is to establish a condition handler.

### STATUS Parameter

SCL commands allow you to specify an optional STATUS parameter having the form:

```
STATUS=variable
```

where variable is a variable of kind status. This variable has three fields, which contain the following information after the command is processed:

The NORMAL field contains the logical value FALSE if errors occurred while the command was executing, and the logical value TRUE if no errors occurred.

The CONDITION field contains a number, known as the condition code, that identifies the error. The condition code has two parts:

- the ASCII representation of a two-character string identifying the processor in control when the error occurred (for example, FC for FORTRAN, CB for COBOL)
- a number that identifies the error for that product

The TEXT field contains message parameters used to substitute into the error message associated with the condition.

After you execute a command, you can test the NORMAL field of the status variable to determine whether an error occurred. Then, on the basis of this test, you can execute statements to perform error processing, exit the block, or perform other desired operations. You can also examine the CONDITION field to identify the particular error.

You reference a particular field of a status variable by specifying

```
status_variable.field.
```

For example, assuming SVAR is a variable of kind status, then the command

```
/display_value  value=svar.normal
```

displays the NORMAL field of SVAR.

If no errors occurred (NORMAL has the value TRUE), the other status variable fields are undefined, and an attempt to reference them results in an error.

There are three important things you must know before specifying a STATUS parameter on an SCL command.

- First, specifying a STATUS parameter on a command automatically suspends normal error processing for that command. Thus, whenever you specify a status parameter, you should be sure to test the status variable. Otherwise, there is no indication that an error occurred.

- Second, before you specify a STATUS parameter on a command, you must first create a variable of kind STATUS through a CREATE_VARIABLE command. You then specify that variable for the STATUS parameter. For example:

```
/create_variable  name=aaa  kind=status
/copy_file  input=infile  output=outfile  status=aaa
```

  In this sequence, a status variable is created and then specified in the STATUS parameter of a COPY_FILE command.

- Third, SCL checks the parameters of each command before SCL executes the command. If the command has an incorrect parameter, the STATUS parameter is not set because SCL never executes the command. For example, entering:

```
/copy_file  input=3_is_not_a_file_name  status=aaa
```

  does not cause the status variable AAA to be set.

The following NOS and NOS/VE examples compile and execute a FORTRAN program. In both examples, a test is made for an error condition following the LGO command. If an error is detected, the error number is displayed. Execution then continues with the command following the ENDIF or IFEND statement.

```
NOS:     FTN5, I=MYPROG.
         SET,EF=0. <-------------------------------- Initialize error flag.
         NOEXIT. <---------------------------------- Suspend normal error processing.
         LGO.
         ONEXIT. <---------------------------------- Restore normal error processing.
         IFE, EF .NE. 0, LAB1.
             DISPLAY, EF.
         ENDIF, LAB1.
```

```
NOS/VE:  create_variable  name=svar  kind=status <-- Define a status variable.
         fortran  input=myprog
         lgo  status=svar <------------------------ Specify the status parameter.  This
                                                    causes the status resulting from
                                                    executing LGO to be returned in the
                                                    status variable SVAR.

         if not svar.normal then <------------------ Test the NORMAL field of the status
                                                    variable.

            display_value  value=svar.condition <-- Display the CONDITION field of the
                                                    status variable.

         ifend
```

### Determining the Condition Value of a STATUS Variable

A condition name is associated with a condition code. Typically you use the condition name to test for a specific condition as in the following example:

```
create_variable  name=pf_status  kind=status
attach_file  file=$user.file_that_doesnt_exist  status=pf_status
if not pf_status.normal  then
    if $condition_name(pf_status.condition) = ..
        'PFE$UNKNOWN_PERMANENT_FILE'
    then
        display_value  'File not in $USER catalog.'
    ifend
ifend
```

The status variable PF_STATUS is set to abnormal status by the ATTACH_FILE command, which makes the NORMAL field of this variable FALSE. The function $CONDITION_NAME returns the condition name from the CONDITION field of the status variable. The condition name is then checked for the specific condition of an unknown permanent file.

### Establishing a Condition Handler

The second method of processing error conditions under NOS/VE is to establish a condition handler. A condition handler is a block of commands that is executed when a specified kind of condition occurs. The simplest form of a condition handler is:

```
WHEN condition DO
    commands
WHENEND
```

If the condition occurs during execution of a command that follows the WHEN/WHENEND block, the commands between WHEN and WHENEND are executed. Control then passes to the command following the command that produced the error.

If the condition does not occur during execution of any commands that follow the WHEN/WHENEND block, the commands between WHEN and WHENEND are not executed.

The following example establishes a condition handler for an LGO command:

```
FORTRAN I=MYPROG
WHEN PROGRAM_FAULT DO
  DISPLAY_VALUE 'An execution error occurred in MYPROG.'
WHENEND
LGO
CANCEL PROGRAM_FAULT
```

If an error, such as a divide fault, occurs while LGO is executing, the DISPLAY_VALUE command is executed. Note that because the FORTRAN command precedes the condition handler, an error during compilation will not cause the condition handler to be executed. The CANCEL statement cancels the condition handler so that it will not be used by subsequent commands.

Remember that specifying a STATUS parameter on a command suspends normal error processing for that command. You can use condition handlers to detect errors in commands that have STATUS parameters.

The ANY_FAULT condition transfers control on the occurrence of any error condition. A useful condition handler to include in a job is as follows:

```
when any_fault do
    put_line ' Command: '//osv$command_name// ..
        ' had abnormal status: '//$strrep(osv$status)
whenend
```

This condition handler displays the command in error and the contents of the variable OSV$STATUS when an error occurs during command processing. OSV$STATUS is a system variable that contains error information.

## Summary of NOS and NOS/VE Execution Control Commands

The following table summarizes the corresponding NOS and NOS/VE execution control commands:

| NOS | NOS/VE |
|---|---|
| WHILE | WHILE/WHILEND (preconditional repetition of commands). |
| - | REPEAT (post-conditional repetition of commands). |
| - | FOR/FOREND (controlled repetition of commands). |
| - | LOOP/LOOPEND (unlimited repetition of commands). |
| IF, IFE, .IF, .IFE | IF |
| EF, EFG symbols | STATUS parameter on commands. |
| EXIT, ONEXIT, NOEXIT | Use STATUS parameter. Also, use condition handlers to test for specific error conditions. |
| ELSE | ELSE |
| SKIP | Use EXIT command. |

# Using Procedures 6

# Using Procedures 6

A procedure is a list of statements that resides on a file. The statements are executed when you call the procedure. Both NOS and NOS/VE enable you to write and call procedures.

Under both systems, you can include any valid system command in a procedure. However, under NOS/VE, the block structure of SCL provides greater flexibility in writing procedures.

This chapter presents an introduction to NOS/VE procedures. Refer to the SCL Language Definition Usage manual for detailed information about SCL procedures.

## Procedure Structure

Under NOS/VE, procedures begin with a PROC statement and end with a PROCEND statement. The statements between PROC and PROCEND constitute the body of the procedure. These statements have the following form:

```
PROC procedure-name (
    parameter 1
    parameter 2
        .
        .
        .
    parameter n
)

    statements
        .
        .
        .
PROCEND procedure_name
```

where procedure-name is a name you assign to the procedure (procedure-name must be a valid SCL name) and parameter 1 to parameter n are parameters (if any).

Under NOS, no statement is required to end the procedure. When the last statement is completed, control simply returns to the system. However, all SCL procedures must be ended by a PROCEND statement. You can omit the name from the PROCEND statement, in which case the PROCEND statement is associated with the nearest preceding PROC statement in the same block.

The following example compares procedures under NOS and NOS/VE. Both procedures display the current date.

```
NOS:      .PROC,DISD.
          DISPLAY,DATE.

NOS/VE:   PROC DISPLAY_DATE
            DISPLAY_VALUE $DATE(MONTH)
          PROCEND
```

In the NOS/VE example, the DISPLAY_VALUE command displays the value of the $DATE function.

NOS/VE provides commands that are similar to the NOS procedure directives. Most of these commands are valid outside as well as inside a procedure. The correspondence between the NOS procedure directives and SCL commands is as follows:

| NOS | NOS/VE |
|---|---|
| .DATA | COLLECT_TEXT |
| .ELSE, ELSE | ELSE, ELSEIF |
| .ENDIF, ENDIF | IFEND |
| .EOF | no equivalent |
| .EOR | no equivalent |
| .EX | no equivalent |
| .HELP/.ENDHELP | no equivalent |
| .IF, .IFE, IF, IFE | IF expression THEN (requires an IFEND statement) The other structured statements can also be used (LOOP, FOR, WHILE, REPEAT). |
| .PROC | PROC (requires a PROCEND statement). |
| REVERT | EXIT_PROC |

The following examples compare the use of some NOS directives with the use of their NOS/VE equivalents. Each procecure receives a single integer argument and determines whether its value is zero, positive, or negative.

NOS:
```
.PROC,TEST*I,A=(*S10(0123456789$-$)).
.IF, A=0 ,TEST1.
     NOTE./THE VALUE IS 0.
.ELSE(TEST1)
.IF, A .LT.  0 ,TEST1.
     NOTE./THE VALUE IS NEGATIVE.
.ELSE(TEST1)
     NOTE./THE VALUE IS POSITIVE.
.ENDIF(TEST1)
REVERT.
```

NOS/VE:
```
proc test (a:integer)
if $value(a) = 0 then
     display_value  value='Value is zero.'
elseif $value(a) < 0 then
     display_value  value='Value is negative.'
else
     display_value  value='Value is positive.'
ifend
procend test
```

The $VALUE function is used to cause parameter substitution in the body of the procedure. This function returns the value of the function argument, in this case, the parameter A. Passing parameters to procedures is discussed later in this chapter.

The following NOS and NOS/VE examples compare the NOS .DATA directive with the NOS/VE COLLECT_TEXT command. Both commands enable you to read data contained in a procedure.

```
NOS:      .PROC,MYPROC.
          FTN5,I=SOURCE,L=FLIST.
          .DATA,SOURCE
          PROGRAM TEST
              .
              .
              .
          END

NOS/VE:   proc myproc
          collect_text  output=source
               program test
                  .
                  .
                  .
               end
          **
          fortran  input=source  list=flist
          lgo
          procend myproc
```

These procedures compile and execute the program contained in file SOURCE. The file is created in the procedure.

## Creating Procedures in NOS/VE

You can create an SCL procedure under NOS/VE using the NOS/VE Full Screen Editor.

## Passing Parameters to Procedures

Both NOS and NOS/VE allow you to write procedures that accept parameters. On both systems, you define the parameters on the procedure header statement. However, the method for defining parameters differs significantly between the two systems. NOS/VE provides greater flexibility in the types of parameters you can define.

Both NOS and NOS/VE prompt for any incorrectly entered parameters and for any required parameters that are missing.

On both systems, the parameter definitions include the following information:

Parameter name

Kinds of values allowed

Default value

Whether or not the parameter is required

Under NOS/VE, parameter definitions on the PROC statement are enclosed in parentheses. Names can have from 1 through 31 characters. Multiple parameters are separated by semicolons. For example, the statement

```
PROC TEST_PROC (P1; P2)
```

defines two parameters, P1 and P2, for procedure TEST_PROC.

The minimum amount of information you must specify in each parameter definition is the parameter name. If you specify no other information, the parameter is assumed to be a file name. Thus, in the preceding example, a caller of procedure TEST_PROC must specify file names for P1 and P2.

As previously mentioned, if you specify no other information in a parameter definition other than the parameter name, the parameter is assumed to be a single file name. You can define other kinds of parameters by including a value specification in the parameter definition. A value specification consists of several clauses. You can use a value specification to:

Define the kind of the parameter (for example, FILE, INTEGER, STRING, BOOLEAN)

Define the parameter to be a variable or an array

Define the parameter to be a list of values or a range of values

## Defining the Parameter Kind

A parameter kind specification appears in a parameter definition as follows:

```
PROC name (parameter_name:kind)
```

Valid kinds are: FILE, NAME, STRING, INTEGER, BOOLEAN, STATUS, and ANY. Some examples are as follows:

| | |
|---|---|
| `PROC TEST (MY_NAME:STRING 10='ME')` | Defines MYNAME to be a STRING parameter 10 characters long, with the default value ME. |
| `PROC MYPROC (INPUT:FILE; OUTPUT:FILE)` | Defines two parameters, named INPUT and OUTPUT, as file names. |

## Defining Parameters to be Variables or Arrays

In addition to passing constant values, such as integers and file names, to procedures, you can pass variables and arrays. The variables and arrays can contain any of the valid value kinds (FILE, INTEGER, STRING, and so forth). For example, using the techniques discussed above, you can write a procedure that is called by the following statement:

```
/myproc  file_name=myfile  nums=15
```

given the following parameter definitions:

```
PROC MYPROC (FILE_NAME:FILE; NUMS:INTEGER)
```

The PROC statement defines two parameters: a FILE parameter and an integer parameter. The parameters must be specified on the call statement as constant values.

You can define a parameter to be a variable or an array by including the VARIABLE or ARRAY clause in the parameter definition, as follows:

```
PROC name (parameter_name:VAR OF kind)
```

or

```
PROC name (parameter_name:ARRAY OF kind)
```

For example, the preceding procedure definition can be changed to accept variables as follows:

```
PROC MYPROC (INPUT_FILE:FILE; NUMS:VAR OF INTEGER)
```

An example of a valid call statement for this procedure is:

```
/j=15
/myproc  input_file=$user.myfile  nums=j
```

where MYFILE is a file in the $USER catalog, and J is a variable of kind INTEGER.

## Defining Parameters to be Lists or Ranges of Values

You can define parameters to accept lists of values or ranges of values. A value list consists of one or more values enclosed in parentheses and separated by commas. For example, the following is a list of integer values:

```
(2,4,6)
```

A range of values is specified by a lower bound and an upper bound separated by two periods. The following example specifies the range of values between 8 and 10, inclusive; that is, the values 8, 9, and 10:

```
8..10
```

To define a parameter to be a list of values, specify the LIST clause in the parameter definition:

```
PROC name (parameter_name:LIST OF kind)
```

For example, the statement

```
PROC MYPROC (PLIST:LIST OF FILE)
```

defines parameter PLIST to be a list of file names.

The following statement defines parameter NUMS to be a list of variables of kind INTEGER:

```
PROC PTEST (NUMS:LIST OF VAR OF INTEGER)
```

To define a parameter to be a range of values, include the RANGE clause in the parameter definition:

```
PROC name (parameter_name:RANGE OF kind)
```

For example, the following statement defines parameter R to be a range of integer values:

```
PROC MPROC (R:RANGE OF INTEGER)
```

An example of a call statement for this procedure is:

```
/mproc r=5..10
```

## Defining Default Values for Parameters

Under both systems, you can define default values for parameters. If a default value is defined, the parameter assumes that value if the procedure user omits the parameter from the call statement. The general form of a default specification under NOS/VE is:

```
PROC name (parameter_name:kind=default)
```

If parameter-name is omitted from the procedure call statement, the parameter assumes the value specified by default. For example, the following statement defines a FILE parameter named INPUT:

```
PROC GGG (INPUT:FILE=SOURCE_FILE)
```

If the FILE parameter is omitted from the statement that calls GGG, the default name SOURCE_FILE is used.

The following examples compare parameter definitions in NOS and NOS/VE procedures:

```
NOS:      .PROC,APROC, P1"FILE TO COMPILE"=(*F,*N=AFILE).
          FTN5,I=P1,L=FLIST.
          LGO.

NOS/VE:   PROC APROC (P1:FILE=AFILE)
            FORTRAN I=$VALUE(P1) L=FLIST
            LGO
          PROCEND
```

In both examples, the name AFILE is defined as the default value for parameter P1. If no parameter is specified on the procedure call, the default value is substituted in the procedure body. For example, commands calling the NOS/VE procedure have the following results:

```
/aproc p1=bfile     File name BFILE is substituted in the FORTRAN command.

/aproc              Default file name AFILE is substituted in the FORTRAN command.
```

## Parameter Substitution

Under NOS, parameter substitution in a procedure is accomplished simply by specifying the parameter name wherever you want the parameter to be substituted. For example:

```
PROC, MYPROC P1=(*F)
ATTACH,P1.
.
.
.
```

When MYPROC is called, the value specified for P1 in the call statement is substituted for P1 in the ATTACH command.

Under NOS/VE, you use the $VALUE function to obtain parameter values. The $VALUE function returns the value of the function argument. Within the procedure body, wherever you want parameter substitution to occur, you place a $VALUE function that names the parameter. For example:

```
PROC ATT (INPUT:FILE)
ATTACH_FILE F=$VALUE(INPUT)
.
.
.
```

When this procedure is called, the value specified for the INPUT parameter is used in the ATTACH_FILE command.

The following example shows parameter substitution for an integer parameter:

```
PROC TEST_PROC (P1:INTEGER)
A = 10 + $VALUE(P1)
DISPLAY_VALUE A
PROCEND
```

When this procedure is called, the $VALUE function returns the value specified for the P1 parameter. If this procedure is called by the statement

```
/test_proc p1=2
```

then the value 12 is displayed.

Another useful function is the $SPECIFIED function, which tells you whether or not a parameter was specified on the procedure call. This function has the form

```
$SPECIFIED(parameter)
```

The $SPECIFIED function returns one of the following values:

TRUE The parameter was specified on the procedure call.

FALSE The parameter was not specified on the procedure call.

The following example illustrates the $SPECIFIED function:

```
PROC MYPROC (INPUT:FILE)
  IF $SPECIFIED(INPUT) THEN
    commands
  ELSE
    commands
  IFEND
PROCEND
```

Procedure MYPROC has a single parameter named INPUT. The IF statement tests the value of the $SPECIFIED function. If the INPUT parameter is specified on the procedure call ($SPECIFIED returns a true value), the first set of commands is processed. If the INPUT parameter is not specified ($SPECIFIED returns a false value), the second set of commands is processed.

Other useful functions include the $VALUE_KIND function, which returns the kind of a parameter, and the $RANGE parameter, which indicates whether or not a parameter is specified as a range.

An important item to remember in writing procedures is that parameter types must match. Consequently, substituting parameters inside your procedure might require the $VALUE function and a conversion function. For example, suppose you want to use the PUT_LINE command to display a parameter of type FILE. The PUT_LINE parameter LINE is of type STRING, so the conversion function $STRREP is used to convert the parameter from type FILE to type STRING. This is illustrated in the following procedure:

```
proc show (
    file_name : file = $required
    )

    put_line  line=' Value for file_name parameter is ' ..
       //$strrep($value(file_name))
procend show
```

### Substituting Parameters in Data Files

NOS CCL permits you to substitute procedure parameters in files created by the .DATA directive, as in:

```
.PROC,SHOW*I,P1=(*N=13).
.DATA,DATAFIL.
VALUE IS P1
```

which creates the file DATAFIL with the contents:

```
VALUE IS 13
```

You can use the COLLECT_TEXT command and its SUBSTITUTION_MARK parameter to do this on NOS/VE. This parameter indicates a character which delimits substitution text. The delimited text is evaluated as a string expression, which then replaces the original text.

The following SCL procedure duplicates the previous CCL proc.

```
proc show (
p1 : integer = 13

)

collect_text  output=datafil ..
    substitution_mark='?' ..
    until='**'
value is ?$strrep($value(p1))?
**

procend show
```

Note that the string expression uses the $VALUE function (to obtain the value of the parameter P1, which is an integer) and then the $STRREP function (to convert the integer to a string). The COLLECT_TEXT command creates the file DATAFIL with the contents:

```
value is 13
```

## Parameter Prompting

Under NOS, to write a procedure that issues input prompts, you append the two characters *I to the procedure name in the header statement. NOS/VE parameter prompting is similar to CCL line mode prompting.

NOS/VE automatically prompts for a procedure's parameters when you enter the following command before calling the procedure:

```
CHANGE_INTERACTION_STYLE  STYLE=style_value
```

style-value can be either LINE (for hard copy terminals) or SCREEN (for video terminals). You should place this command in your user prolog, as described in the SCL System Interface User manual. There is no screen mode parameter prompting.

As with CCL, you use the ? (question mark) to elicit help. Entering ?? (two question marks) results in help on parameter prompting.

## Calling Procedures

Under NOS/VE, the commands to call procedures have the same syntax as other SCL commands. This helps procedure users remember the procedure call commands. NOS/VE provides two ways of calling a procedure. The first way is to create a local file containing a single procedure. You call the procedure by entering the file name. This is similar to the NOS method of calling a procedure by entering its temporary file name, which must be the same as the procedure name. For example, the following sequence creates a temporary file named PROCFILE that contains a procedure named MYPROC:

```
/collect_text output=procfile
ct? procedure myproc
.
.
.
ct? procend myproc
ct? **
```

To execute this procedure, you would enter the file name PROCFILE.

If a procedure is to be executed by specifying its file name, the procedure must be the only procedure on the file.

Both NOS and NOS/VE enable you to store multiple procedures on a single file, and to place the file in an object library. You can then execute any of the procedures simply by entering the procedure name.

On NOS, you use the LIBEDIT utility with the U parameter to add a new procedure to an existing object library:

```
LIBEDIT,P=old-library,B=new-procedure,U,N=new-library.
```

You then use the LIBRARY command to make the library, called a user library, available to the job:

```
LIBRARY,new-library/A
```

On NOS/VE, you use the CREATE_OBJECT_LIBRARY utility to add new procedure(s) to the library and to create the object library. You must then add the procedure names to a list of all commands called the command list.

To create a NOS/VE library that contains procedures, use the following commands:

```
CREATE_OBJECT_LIBRARY
    ADD_MODULES  LIBRARY=file_name
    GENERATE_LIBRARY  LIBRARY=library_name
    QUIT
```

where file_name is a file containing one or more procedures, and library_name is the name of the library being created. The ADD_MODULES directive is like the NOS LIBEDIT utility B parameter; the GENERATE_LIBRARY directive is like the LIBEDIT N and U parameters.

After you create a library containing procedures, you must add the library to the SCL command list.

The SCL command list is a list of commands that the system searches each time you enter an SCL command. In order for the system to find a procedure when you enter the procedure name, the library name must be in the command list. The command to add procedures to the command list is:

```
SET_COMMAND_LIST ADD=library
```

where library is the name of the library that contains the procedures.

In the following NOS/VE example, assume a local file named PROCFILE contains two procedures named PROC1 and PROC2. The file is placed in an object library, and the object library is added to the command list.

```
CREATE_OBJECT_LIBRARY <--------- Begin the CREATE_OBJECT_LIBRARY session.

ADD_MODULES L=PROCFILE <-------- Add procedures from PROCFILE to the library.

GENERATE_LIBRARY L=PROCLIB <---- Generate a library named PROCLIB.

QUIT <------------------------- End the CREATE_OBJECT_LIBRARY session.

SET_COMMAND_LIST ADD=PROCLIB <-- Add the library to the command list.
```

You can now call procedures PROC1 and PROC2 simply by entering their names.

## Displaying the Commands in a Procedure

Commands processed within a procedure are not written to the job log. However, you can display a list of the commands executed in a procedure by creating a connection to the standard file $ECHO.

File $ECHO receives a copy of all commands you enter. It is the closest thing NOS/VE has to the NOS dayfile. File $ECHO is initially connected to file $NULL, which causes the file contents to disappear. By connecting file $ECHO to a local file, you cause the terminal echo to be written to the local file.

The command to connect file $ECHO to a local file is:

```
CREATE_FILE_CONNECTION $ECHO local_file_name
```

By placing this command at the beginning of a procedure, you can cause the commands in the procedure to be written to a local file as they are processed.

The following procedure writes commands to a temporary file as they are processed:

```
proc myproc
create_file_connection  standard_file=$echo  file=echout <-- Connects $ECHO to ECHOUT.
fortran  input=$user.test  list=flist
lgo
delete_file_connection  standard_file=$echo  file=echout <-- Deletes the connection to ECHOUT.
procend myproc
```

After the procedure has executed, you can display the contents of file ECHOUT by entering the command:

```
/copy_file input=echout
```

This command displays the following:

```
CI fortran input=test  list=flist
CI lgo
CI delete_file_connection  standard_file=$echo  file=echout
```

Another useful command for procedures is the DISPLAY_COMMAND_INFORMATION (DISCI) command. This command displays the parameters for any system command and any procedure in your command list. The DISPLAY_COMMAND_INFORMATION command has the form:

```
DISPLAY_COMMAND_INFORMATION  COMMAND=name
```

where name is an SCL command or a procedure name.

The following example shows a procedure definition and a DISPLAY_COMMAND_INFORMATION command:

```
proc myproc (input:file=$input; n:integer=10)

display_command_information  command=myproc
```

```
    input:  file=$input -
    n    :  integer=10  -   Output from DISPLAY_COMMAND_INFORMATION command.
```

## Summary of NOS and NOS/VE Procedure Differences

Procedure directives:

| NOS | NOS/VE |
|---|---|
| .DATA | COLLECT_TEXT (Use SM parameter for substitution.) |
| .ELSE | ELSE, ELSEIF |
| .EOF | no equivalent |
| .EOR | no equivalent |
| .EX | no equivalent |
| .HELP/.ENDHELP | no equivalent |
| .IFE., .IF, IF, IFE | IF expression THEN (requires an IFEND statement). The other structured statements can also be used (LOOP, FOR, WHILE, REPEAT). |
| .ELSE, ELSE | ELSE |
| .PROC | PROC (requires PROCEND at end of procedure). |
| .ENDIF, ENDIF | IFEND |
| REVERT | EXIT_PROC |

Parameter definition:

NOS: .PROC,name,p1,p2,... . Parameters can be numeric or literal strings.

NOS/VE: PROC name (p1:definition,...). Parameters can be constants or variables, and can be lists or ranges of values.

Default value definition:

NOS: .PROC,name,p1=*N=default

NOS/VE: PROC name (p1:kind=default)

Parameter prompting:

NOS: .PROC,name*I,p1,p2,...

NOS/VE: Use CHANGE_INTERACTION_STYLE.

Adding procedures to a library:

NOS: LIBEDIT utility with U parameter.

NOS/VE: CREATE_OBJECT_LIBRARY utility.

Making a library available to a job:

NOS: LIBRARY command adds the library to the global library set.

NOS/VE: SET_COMMAND_LIST command adds the library to the command list.

Calling a procedure on a local file:

NOS: BEGIN,file-name,procedure-name or name call of the file-name only.

NOS/VE: Specify file-name only.

Calling a procedure on a library:

NOS: Specify procedure-name.

NOS/VE: Specify procedure-name.

## Using the NOS/VE Full Screen Editor

You can use the NOS/VE Full Screen Editor (NOS/VE FSE) to create and modify text files.

If you are familiar with the NOS Full Screen Editor (NOS FSE), you should have no difficulty using NOS/VE FSE; the two versions are conceptually similar. However, some command language differences exist; they are discussed below under Differences Between NOS/VE FSE and NOS FSE.

If you are not familiar with NOS FSE, the topics below, starting with Introduction to FSE and ending with Editing a File, provide a general introduction. For more detailed information, see the File Editor Tutorial/Usage manual.

Differences Between NOS/VE FSE and NOS FSE

Introduction to FSE

Beginning and Ending an Editing Session

Entering FSE Commands

Creating a File with FSE

Editing a File

### Differences Between NOS/VE FSE and NOS FSE

The following paragraphs identify some of the significant differences between NOS/VE FSE and NOS FSE. One difference is in the way NOS/VE FSE is invoked. Under NOS, you invoked FSE like this:

```
SCREEN,termtype
FSE,filename
```

Under NOS/VE, you invoke FSE like this:

```
EDIT_FILE FILE=file
ef/
```

This places FSE in line mode. The ef/ is the FSE line mode prompt.

You can set up a startup procedure to be executed each time the EDIT_FILE command is entered. This startup procedure is called a prolog and consists of a series of FSE commands that you place in file $USER.SCU_EDITOR_PROLOG. One command that you typically place in $USER.SCU_EDITOR_PROLOG is the ACTIVATE_SCREEN command, which is described next.

To switch from line mode to full screen mode, enter the ACTIVATE_SCREEN command. For example:

```
ef/activate_screen model=cdc_721
```

The MODEL parameter is required and specifies what model of terminal you are using. The models you can specify include:

CDC_721 (for CDC 721 terminals)

CDC_722 (for CDC 722 terminals)

CDC_722-30 (for CDC 722-30 terminals)

ZEN_Z19 (for Zenith Z19 terminals)

ZEN_Z29 (for Zenith Z29 terminals)

DEC_VT100 (for DEC VT100 terminals)

DEC_VT220 (for DEC VT220 terminals)

If you want to switch back to line mode from full screen mode, position the cursor to the home line and enter:

```
deactivate_screen
```

You can also switch from full screen mode to line mode by entering the terminate-break sequence for your terminal. (Your site administrator knows what this sequence is.)

You need to switch to line mode if you specify the wrong terminal model on the ACTIVATE_SCREEN command. After you enter this kind of ACTIVATE_SCREEN command, you will probably see a wild assortment of characters on the screen. To recover from this situation, enter the terminate-break sequence to return you to line mode. Respond to the ef/ prompt by re-entering the ACTIVATE_SCREEN command, and this time, specify the correct terminal type!

While in FSE, you can enter SCL commands as well as FSE commands. Two SCL commands that are particularly useful are the DISPLAY_COMMAND_LIST command and the DISPLAY_COMMAND_INFORMATION command. These commands provide you with information about other commands, such as the FSE commands.

The SCL DISPLAY_COMMAND_LIST command lists all of the commands that you can enter while in FSE. The resulting display includes all of the NOS/VE FSE commands.

The SCL DISPLAY_COMMAND_INFORMATION displays the parameters of the command you specify. For example:

```
display_command_information command=break_text
```

would display the parameters of the FSE BREAK_TEXT command.

The major difference between NOS/VE FSE and NOS FSE is in the names of the commands. NOS/VE FSE uses the SCL syntax rules and naming conventions.

The following paragraphs provide a summary of the command differences. For a complete description of the NOS/VE file editor commands, see the File Editor Tutorial/Usage manual

| NOS FSE Command | NOS/VE EDIT_FILE Command | Abbreviation(s)/Key |
|---|---|---|
| ALTER | indent_text offset=-integer | INDT/Indent |
| | delete_characters | DELC, DC/DelCh |
| | insert_characters | INSC, IC/InsCh |
| BACK | none | |
| COPY | copy_text | COPT, C/Copy |
| | read_file | REAF/ |
| | write_file | WRIF/ |
| DELETE | delete_characters | DELC, DC/DelCh |
| | delete_lines | DELL, DL/DelLn |
| | delete_text | DELT, D/DelBk |
| DELETE BLANK | delete_empty_lines | DELEL/ |
| DELETE MARK | unmark | /Unmrk |
| DELETE WORD | delete_word | DELW, DW/DelWd |
| FSE | edit_file | EDIF/ |
| FSE SPLIT | set_screen_options split=2; position_cursor .. row=$title_row(2); edit_file | |
| GET ALIGNMENT | display_column_numbers | DISCN/ |
| GET STATUS | display_editor_status | DISES/ |
| HELP | help | H/Help |
| INSERT (I) | insert_lines | INSL, I/InsLn |
| INSERT BLANK | insert_empty_lines | INSEL/ |
| INSERT WORD | insert_word | INSW, IW/InsWd |
| LOCATE (L) | locate_text | LOCT, L/Locate |
| MOVE (M) | move_text | MOVT, M/Move |
| PRINT (P) | position_cursor | POSC, P/ |
| QUIT | quit | Q/Quit |
| QUIT UNDO | quit no | E/Exit |
| REPLACE (R) | replace_text | REPT, R/ |
| SET ANNOUNCE | put_row text='string' row=$message | PUTR/ |
| SET CHAR | set_tab_options character='character' | SETTO/ |
| SET FILENAME | write_file file=file | WRIF/ |
| SET KEY | set_function_key | SETFK/ |
| SET LINE | deactivate_screen | DEAS/ |
| SET MARK | mark_lines | MARL, M/Mark |
| SET MARK WORD | mark_characters | MARC, MC/MrkCh |
| SET PROMPT | set_screen_options menu_row=integer | SETSO/ |
| SET SCREEN | activate_screen | ACTS/ |
| SET TAB | set_tab_options tab_columns=(list of integers) | SETTO/ |
| | clear_tab_options tab_columns=(list of integers) | CLETO/ |
| SET variable | variable=$current_line | |

(The variable in the NOS SET command can be X, Y, or Z. The variable in the NOS/VE EDIT_FILE command can be any SCL variable.)

| NOS FSE Command | NOS/VE EDIT_FILE Command | Abbreviation(s)/Key |
|---|---|---|
| SET VIEW COLUMN | set_screen_options column=integer | SETSO/ |
| SET VIEW EDIT | set_search_margins | SETSM/ |
| SET VIEW OFFSET | align_screen offset=integer | ALIS, A/ |
| SET VIEW WARN | set_line_width | SETLW/ |
| SET WORD CHAR | set_word_characters | SETWC/ |
| SET WORD FILL | set_paragraph_margins | SETPM/ |
| SET WORD variable | variable=$current_column | |
| (The variable in the NOS SET WORD command can be X, Y, or Z. The variable in the NOS/VE EDIT_FILE command can be any SCL variable.) | | |
| SVW;PA | locate_wide_lines | LOCWL/ |
| TEACH | none | |
| UNDO | undo | /Undo |
| UNMARK | unmark | /Unmrk |
| VIEW (V) | align_screen | ALIS, A/ |
| .CENTER | center_lines | CENL/Center |
| .DELETE | delete_characters | DELC, DC/DelCh |
| .END | position_cursor line=current .. column=$strlen($line_text)+1 | POSC, P/ |
| .FILL | format_paragraph | FORP/Format |
| .INSERT | insert_characters | INSC, IC/InsCh |
| .INSERT/'string' | insert_characters new_text='string' | INSC, IC/ |
| .JOIN | join_text | JOIT, J/Join |
| .POS | position_cursor | POSC, P/ |
| .POS integer | position_cursor column=integer | POSC, P/ |
| .SPLIT | break_text | BRET, B/Break |
| -procname | procname | |
| /command | command (does not exit EDIT_FILE) | |
| --comment | "comment | |
| &C | $current_column | |
| &F | $current_object | |
| &L | $current_line | |
| &T | $terminal_model | |
| &W | $current_word | |
| &n | use $value(parameter-name) in the SCL procedure | |
| &? | $screen_input('text') | |
| && | & | |

## Introduction to FSE

You use FSE to create and modify text files, such as programs, program data, source files, and so forth.

FSE operates either in line mode or in screen mode. In screen mode, FSE displays as much text as will fit on a terminal screen. You can edit text anywhere on the screen by moving the cursor to the position where you want to perform the editing operation.

In line mode, you edit single lines of text. Line mode is used mainly with hardcopy terminals. Screen mode is the most powerful mode of editing, and is the subject of the rest of this discussion.

To perform an editing operation on text displayed on the screen, you must move the cursor to the position you want to edit. You control the cursor using the group of four terminal keys labeled with arrows. (The position of these cursor movement keys on the keyboard depends on your terminal type.)

The direction of the arrow indicates the direction the cursor moves when the key is pressed.

Some of the terminals FSE supports are:

CDC 721

CDC 722

Zenith Z19

Digital VT100

Digital VT220

Certain characteristics of FSE might vary among terminal types. For example, the number and position of the keys forming editing operations might differ. In addition, some terminals have keys that allow you to perform certain editing functions independently of FSE. However, the concepts and commands of FSE are identical among terminal types.

## Beginning and Ending an Editing Session

To call FSE, enter the following command:

```
EDIT_FILE file    (abbreviated EDIF)
```

where file specifies the file you want to edit. FSE responds with the following prompt:

```
ef/
```

By default, FSE begins in line mode. To activate screen mode, enter an ACTIVATE_SCREEN (ACTS) command in response to the ef/ prompt, as follows:

```
ef/ACTIVATE_SCREEN  MODEL=termtype
```

where termtype specifies the type of terminal you are using. The following lists some of the possible values of termtype:

| | |
|---|---|
| CDC_721 | for CDC 721 terminals |
| CDC_722 | for CDC 722 terminals |
| ZEN_Z29 | for Zenith Z29 terminals |
| DEC_VT100 | for DEC VT100 terminals |
| DEC_VT220 | for DEC VT220 terminals |

For example, the following commands call FSE and set screen mode:

```
/edit_file  file=myfile  <------------ Call FSE to edit file MYFILE
ef/activate_screen  model=cdc_722  <-- Set screen mode for CDC 722 terminal.
```

After you activate screen mode, FSE displays a screen containing your file (or as much of it as will fit on one screen).

The next screen shows a typical FSE display after you call FSE and enter an ACTIVATE_SCREEN command.

The following is an example of an FSE screen on a CDC 722 terminal:

```
+-------------------------------------------------------------------------------+
|                                                                               |
|       (home line)                                                             |
|                                                                               |
|       (Error message line)                                                    |
|    File:  MYFILE  Lines 19  Thru 37  Size 14675                               |
|                                                                               |
|    The top line of the screen is the Home line.  You enter FSE commands on the Home Line.
|                                                                               |
|    The second line is used by FSE to display error messages.                  |
|                                                                               |
|    The third line tells you whch file you are editing, which of its lines are on the
|    screen, and the size of the file.                                          |
|                                                                               |
|    The middle part of the screen contains the text of your file.              |
|                                                                               |
|    The bottom part of the screen displays keys you can use to perform editing operations.
|                                                                               |
|      First      Last       Copy       Move                Exit      DelCh     DelLn
|   F1 Bkw     F2 Fwd     F3 Back    F4 Help    F5 Undo    F6 Quit    F7 InsCh  F8 InsLn
|                                                                               |
|      Unmrk      MrkBx      LocNxt                                             |
|   F9 Mark    10 MrkCh   11 Locate                                             |
|                                                                               |
+-------------------------------------------------------------------------------+
```

On most terminals, the first line of the FSE display is a blank line called the Home line. On other terminals, the Home line is the last line of the display. In either case, the Home line is where you enter FSE commands, which are described later in this discussion.

FSE displays error messages on the second line. For example, the message:

```
—ERROR CL 790-- COPY is not a command.
```

indicates that the word COPY is not an FSE command.

FSE uses the third line for displaying informative messages about the file you are editing. For example, the message:

```
File:  MYFILE   Lines 19  Thru 37  Size 14675
```

tells that:

You are editing file MYFILE.

The screen is displaying lines 19 through 37 from MYFILE.

MYFILE is 14,675 lines long.

The middle part of the screen contains the text of your file. If the file contains more lines than will fit on a screen, the screen contains the first part of the file. You can move to other parts of the file as explained later in this topic. If you specified a nonexistent file on the EDIT_FILE command, this part of the screen is blank.

The bottom part of the screen shows the FSE commands associated with some keys on your terminal. You can execute any of those commands by pressing the key, as described later in this topic.

To end an editing session, proceed as follows:

1. Press Home. This moves the cursor to the Home line.
2. Press Quit. For some terminals, you also need to press RETURN.

Using Quit terminates FSE, leaving all editing changes intact, and returns control to the operating system.

## Entering FSE Commands

You perform many editing operations by entering FSE commands. You enter FSE commands in one of two ways. The first way is to proceed as follows:

1. Move the cursor to the home line.
2. Enter the desired FSE command.
3. Press RETURN.

The home line is a special line on the terminal screen. The home line is reserved for entering FSE commands. To move the cursor to the home line, press Home.

When you enter a command on the home line and press RETURN, FSE processes the command and returns the cursor to its original position before you pressed Home.

The second, and easier, way of entering FSE commands is to use some keys provided by your terminal. The display at the bottom of the FSE screen shows these keys and their associated commands. For example:

```
   First     Last      Copy      Move                Exit      DelCh     DelLn
F1 Bkw    F2 Fwd    F3 Back   F4 Help   F5 Undo   F6 Quit   F7 InsCh  F8 InsLn

   Unmrk     MrkBx     LocNxt
F9 Mark   10 MrkCh  11 Locate
```

This example shows the display at the bottom of the FSE screen on a CDC 722 terminal. On this terminal, the keys shown are function keys, labeled F1 through 11.

In the display of the editing operations keys, the characters to the right of each key label identify the command that is executed when you press the key. For some terminals you need to also press RETURN to execute the command.

Usually each key has two editing operations associated with it. To perform the bottom operation, press the key. The top operation is usually performed by the shifted key. If a top operation is associated with another key, that key's label is to the left of the operation.

The keys are programmable, so you can change them to suit your own particular needs. They can be programmed to execute any FSE command. If you elect not to program the keys, FSE provides default commands.

Typically, FSE users program the keys to execute the commands they use most often. You might want to use the default functions for awhile, until you become more familiar with FSE. The method for programming the keys is described in the File Editor Tutorial/Usage manual.

The rest of this discussion introduces you to the capabilities of FSE by describing how to do a few simple but useful editing operations. For clarity, the description shows how to do these operations by using the function keys on the CDC 722 terminal. Of course, you can perform these same editing operations and many others by entering FSE commands on the Home line, and you can use FSE on a variety of terminals. To learn the full range of capabilities of FSE, refer to the File Editor Tutorial/Usage manual.

## Creating a File With FSE

To create a new file, enter an EDIT_FILE command that specifies the name of a nonexistent file. Then enter the ACTIVATE_SCREEN command. A screen appears informing you that the file is empty. The text portion of the screen is blank. The cursor is positioned at the top of the screen.

You can now begin entering text into the file. All text you type while the cursor is in the text portion of the screen is entered into the file.

When you reach the end of a line, press RETURN. This positions the cursor at the beginning of the next line, and you can continue typing.

If you reach the end of the screen before you have finished entering information into the file, FSE automatically moves ahead one screen. The last line you typed will appear at the top of the screen, and you can continue typing.

When you have finished entering text into your file, you can leave FSE by using Quit.

If you make a mistake while entering text, you can correct the mistake simply by positioning the cursor where the mistake occurred and typing over the incorrect text.

For example, suppose you have typed the following statement:

```
Ingeter IA(100)
```

You can correct the misspelling of 'Integer' by positioning the cursor under the g as follows:

```
Ingeter IA(100)
  -            <— Cursor position
```

and typing 'teg'.

For many errors, it might not be sufficient to simply type over the error. You may need to delete or insert characters, words, lines, or entire blocks of text. FSE allows you to perform all of these operations, as described in the next topic, Editing a File.

## Editing a File

FSE can perform extensive editing operations on text files. The following topics describe some of the most commonly used operations as used on the CDC 722 terminal. Refer to the File Editor Tutorial/Usage manual for complete descriptions of all the editing capabilities of FSE.

Calling FSE
Paging Through a File
Inserting Characters
Deleting Characters

Inserting a Line
Deleting a Line
Copying a Block of Text
Moving a Block of Text

### Calling FSE

To edit an existing text file, enter an EDIT_FILE command specifying the name of the file. Then enter the ACTIVATE_SCREEN command. FSE displays a screen containing as much of your file as will fit on the screen, beginning with the first line of the file. For example, the following commands call FSE to edit a file named MYFILE and activate screen mode for a CDC 722:

```
/edit_file myfile
ef/activate_screen model=cdc_722
```

The FSE display for this example is shown below:

```
File:  MYFILE  Lines 1  Thru 7  Size 7

   Program MYPROG
   Integer IA(100)
   DO 10 I=1,100
     IA(I) = I
10 Continue
   Print *, ' End of Program MYPROG'
   End

   First      Last         Copy       Move                 Exit       DelCh      DelLn
F1 Bkw    F2 Fwd       F3 Back    F4 Help    F5 Undo    F6 Quit    F7 InsCh   F8 InsLn

   Unmrk      MrkBx        LocNxt
F9 Mark   10 MrkCh     11 Locate
```

The top line displays the name of the file being edited. The middle part of the screen displays the contents of the file, and the bottom of the screen displays the keys used for editing operations.

To page ahead one screen in the file, use Fwd. The last line on the previous screen becomes the first line on the new screen.

To page backward one screen in the file, use Bkw. The first line on the previous screen becomes the last line on the new screen.

To move to the end of the file, use Last.

To move to the beginning of the file, use First.

### Inserting Characters

To insert one or more characters anywhere in your text file, proceed as follows:

1. Position the cursor to the character after which you want to insert the new characters.

2. Press InsCh once for each character you want to insert.

For each press of InsCh, FSE inserts a blank character immediately to the right of the cursor position. You can then position the cursor anywhere within the field of blanks and type the desired characters.

In the following example, the character t is inserted into the word Ineger.

1. Move the cursor to the position immediately preceding the position of the insert.

```
Program MYPROG
Ineger A(100)
 -            <-- cursor position
```

2. Press InsCh.

```
Program MYPROG
In eger A(100)
  -
```

3. Type the desired character.

```
Program MYPROG
Integer A(100)
  -
```

The following example shows how to insert a string of characters. The string 'B(100), ' is inserted between Integer and A(100).

1. Move the cursor to the position after which the string is to be inserted.

```
Program MYPROG
Integer A(100)
        -       <-- cursor position
```

2. Press InsCh 8 times (corresponding to the 7 characters in 'B(100),' plus one blank).

```
Program MYPROG
Integer         A(100)
        -
```

3. Type the new string.

```
Program MYPROG
Integer B(100), A(100)
                -
```

### Deleting Characters

The following example shows how to delete a string of characters. The string 'IB(100), ' is deleted from the line 'Integer IB(100), IA(100)'.

1. Move the cursor to the first character of the string to be deleted.

```
Program MYPROG
Integer IB(100), IA(100)
        -       <-- cursor position
```

2. Press DelCh 8 times (corresponding to the 7 characters in 'IB(100)', plus one blank).

```
Program MYPROG
Integer IB(100), IA(100)
                -
```

3. FSE deletes 8 characters.

```
Program MYPROG
Integer IA(100)
        -
```

### Inserting a Line

The following example shows how to insert a line of text. The line 'Real A(100)' is inserted between the lines 'Program MYPROG' and 'Integer IA(100)'.

1. Move the cursor to the line above which the new line is to be inserted.

```
Program MYPROG
Integer IA(100)
 -              <-- cursor position
```

2. Press InsLn. FSE inserts a blank line above the current line.

```
Program MYPROG

Integer IA(100)
```

3. Type the new line.

```
Program MYPROG
Real A(100)
Integer IA(100)
```

### Deleting a Line

The following example shows how to delete a line of text. The line Real A(100) is deleted.

1. Move the cursor to the line to be deleted.

```
Program MYPROG
Real A(100)
-               <-- cursor position
Integer IA(100)
```

2. Press DelLn. FSE deletes the line at the current cursor position.

```
Program MYPROG
Integer IA(100)
```

### Copying a Block of Text

To copy one or more text lines from one area of the file to another, you use Mark and Copy.

1. Move the cursor to the first line of text to be copied.
2. Press Mark. This marks the line at the current cursor position.
3. Move the cursor to the last line of text to be copied.
4. Press Mark. This marks all lines following the first marked line down to and including the line at the current cursor position.
5. Move the cursor to the line before which you want the text to be copied.
6. Press Copy. This copies the marked lines to the current cursor position.

In the following example, two COMMON statements in a main program are copied to a subroutine.

1. Move the cursor to the first line of text to be copied. Then press Mark.

```
Program Main
Common A(100), B(100)
-                    <-- Cursor position
Common C(100), D(100)
.
.
End

Subroutine Sub
Dimension J(10)
.
.
End
```

2. Move the cursor to the last line of text to be copied. Then press Mark.

```
Program Main
Common A(100), B(100) }
Common C(100), D(100) } <-- marked lines
-                       <-- cursor position
.
.
End

Subroutine Sub
Dimension J(10)
.
.
End
```

3. Move the cursor to the line before which you want the text to be moved.

```
Program Main
Common A(100), B(100) }
Common C(100), D(100) } <-- marked lines
.
.
End

Subroutine Sub
Dimension J(10)
-                       <-- cursor position
.
.
End
```

4. Press Copy.

```
Program Main
Common A(100), B(100)
Common C(100), D(100)
.
.
End

Subroutine Sub
Common A(100), B(100) }
Common C(100), D(100) } <-- copied lines
Dimension J(10)
-                       <-- cursor position
.
.
End
```

### Moving a Block of Text

To move one or more text lines from one area of a file to another, you use Mark and Move.

1. Move the cursor to the first line of text to be moved.
2. Press Mark. This marks the line at the current cursor position.
3. Move the cursor to the last line of text to be moved.
4. Press Mark. This marks all lines following the first marked line down to and including the line at the current cursor position.

5. Move the cursor to the line before which you want the text to be moved.

6. Press Move. The marked lines are moved to the current cursor position.

In the following example, two COMMON statements in a main program are moved to a subroutine.

1. Move the cursor to the first line of text to be moved. Then press Mark.

```
Program Main
Common A(100), B(100)
-                       <-- cursor position
Common C(100), D(100)
.
.
End

Subroutine Sub
Dimension J(10)
.
.
End
```

2. Move the cursor to the last line of text to be moved. Then press Mark.

```
Program Main
Common A(100), B(100) }
Common C(100), D(100) } <-- marked lines
-                       <-- cursor position
.
.
End

Subroutine Sub
Dimension J(10)
.
.
End
```

3. Move the cursor to the line before which you want the text to be moved.

```
Program Main
Common A(100), B(100) }
Common C(100), D(100) } <-- marked lines
.
.
End

Subroutine Sub
Dimension J(10)
-                       <-- cursor position
.
.
End
```

4. Press Move.

```
Program Main
.
.
End

Subroutine Sub
Common A(100), B(100) }
Common C(100), D(100) } <-- moved text
Dimension J(10)
-                       <-- cursor position
.
.
End
```

# Compiling, Loading, and Executing Programs 7

# Compiling, Loading, and Executing Programs 7

This chapter discusses the commands and techniques for compiling, loading, and executing FORTRAN and COBOL programs under NOS/VE and compares them with the methods used under NOS.

## Compiling FORTRAN Programs

The command to compile a FORTRAN program under NOS/VE is:

```
FORTRAN parameter-list
```

The FORTRAN command follows the same syntax rules as other SCL commands. Parameters have the form

```
parameter=value
```

Parameters are separated by a comma or a space.

Examples of comparable NOS and NOS/VE compiler commands are:

```
NOS:     ftn5,i=sfile,b=bfile,l=lfile.

NOS/VE:  fortran input=sfile binary_object=bfile list=lfile

         ftn i=sfile b=bfile l=lfile
```

All three commands read source input from file SFILE, write object code to file BFILE, and write an output listing to file LFILE.

The third of the preceding examples shows the short form of the NOS/VE FORTRAN command. The FORTRAN command can be abbreviated to FTN. In addition, most of the parameters have a long form and an abbreviation. For example, the following commands are equivalent:

```
/fortran input=srcfile binary_object=binfile list=listfile

/ftn i=srcfile b=binfile l=listfile
```

The FORTRAN command parameters differ from those of FTN5. Some of the parameters are equivalent but have different names, some new parameters have been added, and some old parameters have been dropped.

The following tables present a quick comparison of the NOS and NOS/VE FORTRAN parameters. The tables show the parameter names and a short description of each parameter. The abbreviations of the NOS/VE parameters are shown in parentheses. The first table shows the most common NOS and NOS/VE parameters, the second table shows the rest of the corresponding NOS and NOS/VE parameters, the third table shows new NOS/VE FORTRAN parameters, and the last table shows NOS FTN5 parameters that are not supported under NOS/VE. For complete descriptions of all of the FORTRAN parameters, refer to chapter 14, Migrating FORTRAN Programs.

Following are the most commonly used parameters. Note that the NOS/VE abbreviations are identical to the NOS parameter names.

| NOS | NOS/VE | Description |
|---|---|---|
| B | BINARY_OBJECT (B) | File to receive object code. |
| I | INPUT (I) | File containing input source code. |
| L | LIST (L) | File to receive output listing. |
| LO | LIST_OPTIONS (LO) | Output listing options. |

Following are the rest of the corresponding NOS and NOS/VE FORTRAN parameters. The parameters are equivalent, but in some cases the names have changed.

| NOS | NOS/VE | Description |
|---|---|---|
| ANSI | STANDARDS_DIAGNOSTICS (SD) | Diagnose non-ANSI usages. |
| CS | DEFAULT_COLLATION (DC) | Specifies default collating sequence. NOS default is FIXED, NOS/VE default is USER. |
| DB | DEBUG_AIDS (DA) | Generates symbol tables for interactive debugging. NOS default is all tables generated, NOS/VE default is no tables generated. |
| DO | ONE_TRIP_DO | Minimum trip count for DO loops. |
| DS | COMPILATION_DIRECTIVES (CD) | Suppresses C$ compiler directives. |
| E | ERROR (E) | File to receive error messages. |
| EL | ERROR_LEVEL (EL) | Severity level of messages written to ERROR file. |
| ET | TERMINATION_ERROR_LEVEL (TEL) | Severity level for which abnormal status returned. |
| MD | MACHINE_DEPENDENT (MD) | Requests diagnosis of machine dependencies. |
| OPT | OPTIMIZATION_LEVEL (OPTIMIZATION, OPT, OL) | Compiler optimization level. NOS default is 0, NOS/VE default is LOW. |
| SEQ | SEQUENCED_LINES (SL) | Selects sequenced source format. |
| TM | TARGET_MAINFRAME (TM) | Not supported in this release, but occupies a position in the FORTRAN command. |

The following parameters have been added to the NOS/VE FORTRAN command:

| Parameter | Description |
|---|---|
| EXPRESSION_EVALUATION (EE) | Selects the order of evaluation of expressions. |
| FORCED_SAVE (FS) | Saves variables and arrays in subprograms. |
| RUNTIME_CHECKS (RC) | Specifies runtime range checking of subscript substring expressions. Default is no checking. |
| STATUS | Specifies an SCL status variable to receive an error status code. |

The following NOS FTN5 parameters are not supported under NOS/VE FORTRAN:

| | |
|---|---|
| AL | Not needed with virtual memory. |
| ARG | Not needed because all languages have common calling sequences. |
| BL | Controlled by PAGE_FORMAT file attribute set by SET_FILE_ATTRIBUTE command. |
| EC | Not needed with virtual memory. |
| G | Not needed because COMPASS not available. |
| GO | Not available. |

| | |
|---|---|
| LCM | Not needed with virtual memory. |
| ML | Not needed because COMPASS not available. |
| PD | Not available. |
| PL | Not available. |
| PN | Not available. |
| PS | Controlled by PAGE_LENGTH file attribute set by SET_FILE_ATTRIBUTE command. |
| PW | Controlled by PAGE_WIDTH file attribute set by SET_FILE_ATTRIBUTE command. |
| QC | Not available. |
| REW | Not needed. All files rewound by default. Other file positioning available via NOS/VE file referencing. |
| ROUND | Not available. |
| S | Not needed. |
| STATIC | Not available. |
| X | Not needed. |

## Compiling COBOL Programs

Under NOS/VE, you compile COBOL programs by using the COBOL command. This command has the following form:

```
COBOL parameter_list
```

The COBOL command follows the same syntax rules as other SCL commands. Parameters have the form

```
parameter=value
```

Parameters are separated by a comma or a space.

Examples of comparable NOS and NOS/VE compiler commands are:

```
NOS:     COBOL5,I=SFILE,B=BFILE,L=LFILE.

NOS/VE:  cobol input=sfile binary=bfile list=lfile
```

Both commands read source input from file SFILE, write object code to file BFILE, and write an output listing to file LFILE.

Most of the COBOL command parameters have a long form and an abbreviation. For example, the following commands are equivalent:

```
/cobol input=srcfile binary=binfile list=listfile

/cobol i=srcfile b=binfile l=listfile
```

The COBOL command parameters differ from those of COBOL5. Some of the parameters are equivalent but have different names, some new parameters have been added, and some old parameters have been dropped.

The following tables present a quick comparison of the NOS and NOS/VE COBOL parameters. The tables show the parameter names and a short description of each parameter. The abbreviations of the NOS/VE parameters are shown in parentheses. The first table shows the most commonly used parameters, the second table shows the rest of the corresponding NOS and NOS/VE parameters, the third table shows new NOS/VE COBOL parameters, and the last table shows NOS COBOL5 parameters that are not supported under NOS/VE. For complete descriptions of all of the COBOL parameters, refer to chapter 15, Migrating COBOL Programs.

Following are the most commonly used COBOL command parameters. Note that the NOS/VE abbreviations are identical to the NOS parameter names.

| NOS | NOS/VE | Description |
|---|---|---|
| B | BINARY (B) | File to receive object code. |
| I | INPUT (I) | File containing input source code. |
| L | LIST (L) | File to receive output listing. |
| LO | LIST_OPTIONS (LO) | Output listing options. |

Following are the rest of the corresponding NOS and NOS/VE parameters. The parameters are equivalent, but in some cases the names have changed.

| NOS | NOS/VE | Description |
|---|---|---|
| ANSI | STANDARDS_DIAGNOSTICS (SD) | Diagnose non-ANSI usages. |
| APO | LITERAL_CHARACTER (LC) | Specify character (' or ") to denote literals. |
| B | BINARY (B) | File to receive object code. |
| DB | DEBUG_AIDS (DA) | Debugging options. |
| E | ERROR (E) | File to receive error messages. |
| EL | ERROR_LEVEL (EL) | Severity level of messages to be written to ERROR file. |
| FIPS | FED_INFO_PROCESSING_STANDARD (FIPS) and STANDARDS_DIAGNOSTICS (SD) | Diagnose non-FIPS usages. |
| I | INPUT (I) | Source input file. |
| L | LIST (L) | Output listing file. |
| LBZ | LEADING_BLANK_ZERO (LBZ) | Leading blanks in numeric fields interpreted as zeros. |
| LO | LIST_OPTIONS (LO) | Output listing options. |
| OPT | OPTIMIZATION_LEVEL | Controls compiler optimization level. |
| SB | SUBPROGRAM (SP) | Compile source input as subprogram. |
| SY | DEBUG_AIDS (DA) | Check syntax but do not produce object code. |
| X | EXTERNAL_INPUT (EI) | Specifies SCU library file to be used for COPY statements. |

The following new parameters have been added to NOS/VE COBOL:

| Parameter | Description |
|---|---|
| AUDIT (A) | Selects Federal Compiler Testing Center audit testing. |
| BASE_LANGUAGE (BL) | Allows NOS/VE to compile programs containing syntax based on different base languages. |
| INPUT_SOURCE_MAP (ISM) | Specifies the name of the file that contains the source map describing the contents of the source input file. |
| RUNTIME_CHECKS (RC) | Selects runtime checking of reference modifiers, subscripts, and index references. |
| STATUS | Specifies an SCL status variable to receive error status code. |

The following COBOL5 parameters are not supported under NOS/VE COBOL:

| | |
|---|---|
| BL | Use the PAGE_FORMAT file attribute set by the SET_FILE_ATTRIBUTE command. |
| CC1 | |
| D | |
| ET | Use TERMINATION_ERROR_LEVEL attribute set by SET_PROGRAM_ATTRIBUTE command. |
| FDL | |
| MSB | Not needed. |
| PD | |
| PS | Use the PAGE_LENGTH file attribute set by the SET_FILE_ATTRIBUTE command. |
| PSQ | |
| PW | Use the PAGE_WIDTH file attribute set by the SET_FILE_ATTRIBUTE command. |
| SORT4 | |
| SORT5 | |
| TAF | |
| TDF | |
| U | |
| UC1 | |

# Compiling Pascal Programs

Both NOS and NOS/VE provide a PASCAL command to call the Pascal compiler. Both commands read a Pascal source program and produce an object program that can be loaded and executed. However, the formats of the commands, and the compile time options that can be selected, differ between the two systems.

## Comparison of NOS and NOS/VE PASCAL Commands

The command to compile a NOS/VE Pascal program has the following general format:

```
pascal  input=srce  binary=bin  list=lfile
```

where parameter-list selects various compiler options. The PASCAL command conforms to SCL command syntax, which differs from NOS syntax.

Parameters on the PASCAL command have the following form:

```
parameter_name=value
```

Following are examples of equivalent NOS and NOS/VE PASCAL commands:

```
NOS:     PASCAL,I=SRCE,B=BIN,L=LFILE.

NOS/VE:  pascal input=srce binary=bin list=lfile
```

Both commands compile a source program on local file SRCE, write object code to file BIN, and write the output listing to file LFILE. Note that the NOS/VE PASCAL command uses spaces as parameter separators (although commas can also be used), and does not require a period terminator.

Parameter names on the PASCAL command have abbreviations that can be substituted for the full name. Using parameter abbreviations, the above command can be written as:

```
/pascal i=srce b=bin l=lfile
```

Following are parameters that are provided by both the NOS and NOS/VE PASCAL commands. The NOS/VE parameter abbreviations are shown in parentheses. Note that the NOS/VE abbreviation is the same as the NOS parameter name.

| NOS | NOS/VE | Remarks |
|---|---|---|
| I | INPUT (I) | Source input file. Default is INPUT on NOS, $INPUT on NOS/VE, with similar effect. |
| B | BINARY (B) | Binary object file. Default is LGO on NOS and NOS/VE. |
| L | LIST (L) | Output listing file. Default is OUTPUT on NOS, and $LIST on NOS/VE with similar effect. |

The following NOS PASCAL parameters are not provided by NOS/VE PASCAL:

GO

PD

PS (Page size is controlled by the PAGE_WIDTH file attribute.)

PL

The following new parameters are provided by NOS/VE PASCAL:

| | |
|---|---|
| DEBUG_AIDS | Requests compiler-generated debugging tables. |
| ERROR | Specifies a file to receive error messages. |
| ERROR_LEVEL | Specifies severity level of errors to be listed. |
| LIST_OPTIONS | Selects output listing options. |
| OPTIMIZATION_LEVEL | Selects optimization level. |
| RUNTIME_CHECKS | Selects runtime error checking options. |
| STANDARDS_DIAGNOSTICS | Selects compiler diagnosis of nonstandard usages. |
| TERMINATION_ERROR_LEVEL | Specifies severity level for which compiler is to return abnormal status. |
| STATUS | Specifies a status variable to receive status code upon completion of compilation. |

## Loading and Executing Programs

Under both NOS and NOS/VE, a compiled object program must be processed by the system loader before it can be executed.

### The Loading Process: NOS and NOS/VE Differences

Internally, the loading process differs significantly between NOS and NOS/VE. Externally, however, the processes are similar. Under both systems, the loader performs the following sequence of events:

Allocates an area of memory for the program.

Loads the program into the allocated area.

Loads all routines required by the loaded program. (The loader searches the available libraries for the required routines.)

Supplies all calling programs with the addresses of the called routines. (This process is known as satisfying external references.)

Begins execution of the loaded program.

Many of the loader options provided by NOS/VE are similar to those provided by NOS. Under both systems you can:

Load routines from a single file.

Load routines from multiple files.

Load routines from libraries.

Request a load map.

Under NOS/VE, the overlay, OVCAP (overlay capsule), and segment loading features are not provided. Because of the virtual memory design of the NOS/VE operating system, those features are no longer necessary.

Virtual memory enables you to execute very large programs in a limited amount of physical memory, as though the amount of physical memory was unlimited. Virtual memory operates automatically; it requires no changes to programs, and no special instructions or intervention from the programmer. However, program size and program design have a significant effect on the performance of a program in virtual memory.

## Name Call Loading

The commands to perform a name call load are the same under NOS and NOS/VE. Under both systems you simply specify the object file name. The loader loads all modules from the specified file into memory and begins execution.

Under both systems, language compilers write object code to file LGO if no object file name is specified for the BINARY_OBJECT or BINARY parameter on the compiler call command. Thus, the familiar execution command LGO is the same on both systems.

Both systems allow you to specify parameters on the name call command, which are passed to the execution program. Some of the parameters are similar, while others differ greatly. Parameters are discussed later in this chapter.

The following examples illustrate name call loads on NOS and NOS/VE:

```
NOS:      FTN, I=MYSOURCE.
          LGO.

NOS/VE:   fortran input=mysource
          lgo
```

In the preceding examples, object code is written to file LGO. The LGO command loads and executes the program.

```
NOS:      COBOL5,I=COBPRG,B=BIN.
          BIN.

NOS/VE:   cobol input=cobprg binary=bin
          bin
```

In the preceding examples, object code is written to file BIN. The BIN command loads and executes the program.

## Using EXECUTE_TASK Instead of Load Sequences

Under both systems, the name call command provides a set of default load-time options. However, the method for requesting additional options, or overriding the default options, differs between the two systems.

Under NOS, you used load sequences to request load operations. A load sequence consisted of a sequence of loader control statements terminated by a loader termination statement (EXECUTE, NOGO, or a name call). The loader statements specified various loader options.

Under NOS/VE, you initiate a load operation through the EXECUTE_TASK (EXET) command. This command has the general form:

```
EXECUTE_TASK parameter-list  (abbreviated EXET)
```

The parameter list requests the desired loader options. This command has the same general purpose as a NOS load sequence: It calls the loader, requests loader options, and begins execution of the loaded program.

The following example shows a typical load operation under NOS and NOS/VE:

```
NOS:      LOAD (BIN1, BIN2)
          LDSET (MAP=ON)
          EXECUTE.

NOS/VE:   execute_task files=(bin1,bin2) load_map_options=(s,b)
```

These examples load all modules from object files BIN1 and BIN2, produce a load map, and begin execution of the loaded program.

Under NOS/VE, you can also request certain loader and execution-time options with the SET_PROGRAM_ATTRIBUTES command. The difference between specifying options in an EXECUTE_TASK command and specifying them in a SET_PROGRAM_ATTRIBUTES command is this: the former command establishes the options only for the current execution; the latter establishes them for the duration of the job (unless you turn the options off before the end of the job).

For example, you can turn on the load map option for the duration of the job with the command

```
/set_program_attributes load_map_options=(s,b)
```

After you enter this command, all subsequent load operations will produce a load map (unless you first turn the load map option off).

The following list shows corresponding NOS and NOS/VE loader options. NOS options are selected by loader control statements placed in a load sequence. NOS/VE options are selected by parameters on the EXECUTE_TASK command.

Name Call Load:

NOS: Specify object file name (default is LGO).

NOS/VE: Specify object file name (default is LGO).

Loading from Multiple Files:

NOS: LOAD statement.

NOS/VE: FILE= parameter on EXET command.

Loading from Libraries:

NOS: LIBLOAD statement.

NOS/VE: MODULES= parameter on EXET command or SETCL command.

Begin Execution:

NOS: EXECUTE or name call command.

NOS/VE: EXECUTE_TASK initiates execution after load is complete.

LDSET Options:

| NOS | NOS/VE |
|---|---|
| MAP option. | LOAD_MAP and LOAD_MAP_OPTIONS parameters on the EXET command. |
| LIB option. | LIBRARY= parameter on the EXECUTE_TASK command. |
| PRESET, PRESETA options | PRESET_VALUE= parameter on EXECUTE_TASK command. |
| LIBLOAD option | LIBRARY= parameter on EXECUTE_TASK command. |
| ERR option | TERMINATION_ERROR_LEVEL= parameter on EXECUTE_TASK command. |

## Passing Parameters to a Program

NOS/VE enables you to pass parameters to a program through the name call or EXECUTE_TASK command.

### Passing Parameters to a FORTRAN Program

NOS/VE FORTRAN allows you to pass the following types of parameters to a program:

Predefined parameters

File substitution parameters

User-defined parameters

The predefined parameters are the $PRINT_LIMIT and STATUS parameters. The $PRINT_LIMIT parameter is similar to the NOS *PL parameter. It specifies the maximum number of lines that can be written to files $ERRORS and $OUTPUT during execution.

The STATUS parameter specifies a variable of kind status that receives an error status code at the completion of execution. The format of the information contained in the status variable has the same format as for all other SCL commands. The SCL STATUS parameter is described in chapter 5, Job Structure.

The execution-time file substitution parameters under NOS/VE have the same format as under NOS. The formats of these parameters are described in the FORTRAN usage manual.

The user-defined parameters allowed under NOS/VE FORTRAN differ significantly from those allowed under NOS. Under NOS/VE, user-defined parameters on the name call or EXECUTE_TASK command have the same format as for SCL procedures. SCL parameter formats are briefly described in chapter 6, Using Procedures.

The NOS GETPARM call for retrieving the values of parameters specified on the execution call command is replaced by the NOS/VE parameter and variable interface calls. Brief descriptions of these calls are presented in chapter 14, Migrating FORTRAN programs.

The NOS Post Mortem Dump parameters are not supported under NOS/VE. (A similar capability is provided by the SCL ABORT_FILE capability, which allows you to specify a file of debugging commands that is executed automatically when a runtime error occurs.)

### Passing Parameters to a COBOL Program

Under NOS, you can specify the following parameters on an execution command: *CORE, *MSGS, *TIME, and file equivalencing parameters.

Under NOS/VE, the *CORE, *MSGS, and *TIME parameters are not supported. (The *TIME parameter placed the program's execution time in the job dayfile. Under NOS/VE, the execution time is automatically placed in the job log.) File equivalencing is supported under NOS/VE and has the same format as under NOS.

NOS/VE also provides an additional parameter called the STATUS parameter. The STATUS parameter specifies an SCL status variable in which an error status code is returned after execution completes. The format of the information returned in the status variable is the same as for other SCL commands. STATUS variables are discussed in chapter 5, Job Structure.

## Loading Programs From Multiple Files

Both NOS and NOS/VE allow you to load modules from several different files. Under NOS, you use the LOAD statement to specify files from which modules are to be loaded. Under NOS/VE, the FILES parameter on the EXECUTE_TASK command specifies a list of files from which modules are to be loaded. The FILES parameter has the form:

```
EXECUTE_TASK FILES=(file1, ..., filen)
```

All modules from the specified files are loaded into a single program in memory. Execution begins with the main program in the first file in the list.

NOS/VE does not provide an operation equivalent to the SLOAD operation on NOS, which loads selected modules from a specified file. However, you can load selected modules from a library file, as described later in this chapter.

Some NOS and NOS/VE examples are as follows:

```
NOS:     LOAD(BINFIL1, BINFIL2, LGO)
         EXECUTE.

NOS/VE:  execute_task files=(lgo, binfil1, binfil2)
```

These commands load and execute the modules on files LGO, BINFIL1, and BINFIL2.

## Loading Programs From Libraries

Both NOS and NOS/VE make use of libraries containing object programs. Under NOS, they are called user libraries; under NOS/VE, they are called object libraries. This discussion explains how the NOS/VE loader uses object libraries. Library creation is discussed later in this chapter.

Under both systems, at the end of a load operation, the loader searches the available libraries for modules needed to satisfy external references in the loaded program. On finding a required module, the loader loads it into memory and satisfies the external reference. Under NOS, the loader searches three sets of libraries, in the order listed:

Global library set (established by LIBRARY statement)

Local library set (established by LDSET statement. Also included product libraries, such as FTNLIB or COB5LIB)

System default library (SYSLIB)

Under NOS/VE, immediately before the library search, the loader constructs a list of libraries called the program library list. The loader then searches this list to satisfy external references. The program library list consists of the following lists:

1. Local library list. Available only to the current load operation. (Similar to the local library set under NOS.)

2. Object libraries specified by the compiler. These include such libraries as the FORTRAN and COBOL runtime libraries.

3. Job library list. Available to all load operations in the current job. (Similar to the global library set under NOS.)

4. Job debug library list. Includes libraries used by the Debug utility. (Used only when debug mode is on or when Debug gets control through an ABORT_FILE).

5. NOS/VE task services library. This is a system default library that contains routines required by most application programs.

6. CYBIL Library. Contains routines used by all programs.

The program library list, searched by the loader to satisfy external references, is made up of the preceding lists. You can use parameters on the EXECUTE_TASK and SET_PROGRAM_ATTRIBUTES commands to add libraries to these lists.

The LIBRARY parameter on the EXECUTE_TASK command adds libraries to the local library list:

```
EXECUTE_TASK FILES=(file1,...,filen) LIBRARY=(library1,...,libraryn)
```

The local library list is available only to the EXECUTE_TASK command in which it is specified. The local library list is searched before any other libraries. Libraries in the local library list are searched in order of their occurrence in the list.

The ADD_LIBRARY parameter on the SET_PROGRAM_ATTRIBUTES command adds libraries to the job library list:

```
SET_PROGRAM_ATTRIBUTES ADD_LIBRARY=(library1,...,libraryn)
```

The job library list is available to all subsequent load operations in the job. When you specify a SET_PROGRAM_ATTRIBUTES command, the libraries are added to the beginning of the list.

You can delete libraries from the job library list by specifying the DELETE_LIBRARY parameter of the SET_PROGRAM_ATTRIBUTES command:

```
SET_PROGRAM_ATTRIBUTES DELETE_LIBRARY=(library1,...,libraryn)
```

The following examples assume that debug mode is off (no debug library list) and that the job library list is initially empty.

| Commands | Order of Library Search |
|---|---|
| lgo | Compiler-supplied libraries<br>Task services library |
| exet file=lgo library=(alib, blib) | ALIB<br>BLIB <-- Local libraries<br>Compiler-supplied libraries<br>Task services library |
| setpa add_library=clib<br>exet file=lgo library=(alib, blib) | ALIB<br>BLIB <-- Local libraries<br>Compiler-supplied libraries<br>CLIB <-- Job library<br>Task services library |

Both NOS and NOS/VE enable you to load selected modules from object libraries. Under NOS, you use the LIBLOAD statement to specify the library and the modules to be loaded. The modules are loaded regardless of whether or not they are needed to satisfy external references.

Under NOS/VE, the MODULES parameter of the EXECUTE_TASK command performs the equivalent operation. The MODULES parameter has the following form:

```
EXECUTE_TASK FILES=(file1,...,filen) MODULES=(module1,...,modulen)
```

This command first loads all modules from the files specified by the FILES parameter. It then loads the modules specified by the MODULES parameter from the program library list (the list of libraries that the loader searches during a load operation) regardless of whether or not they are needed to satisfy external references.

The following NOS and NOS/VE examples illustrate loading from libraries:

```
NOS:      LIBLOAD (MYLIB, SUB1, SUB2).
          LGO.

NOS/VE:   execute_task files=lgo modules=(sub1, sub2)
```

In both of these examples, a program is loaded from file LGO. In the NOS example, modules SUB1 and SUB2 are loaded from library MYLIB. In the NOS/VE example, SUB1 and SUB2 are loaded from the program library list.

## Generating a Load Map

Both NOS and NOS/VE provide an optional load map. The load map shows how the loader allocated memory during the load operation. Both systems provide similar types of information on their load maps (module length, module addresses, entry point addresses, entry point cross reference list). However, the methods of interpreting the information on the map differ between the two systems, because the way memory is allocated under NOS/VE differs from that of NOS. For a complete description of the content of a NOS/VE load map, refer to the SCL Object Code Maintenance Usage manual.

Under NOS/VE, you can request a load map either through the SET_PROGRAM_ATTRIBUTES command or through the EXECUTE_TASK command. The system default is to not produce a load map. If you request a load map, it is written to the default file $LOCAL.LOADMAP unless you specify a different file.

The first way of requesting a load map is to specify the LOAD_MAP and LOAD_MAP_OPTIONS parameters on the EXECUTE_TASK command:

```
EXECUTE_TASK FILES=files LOAD_MAP=file LOAD_MAP_OPTIONS=(option_list)
```

The LOAD_MAP parameter specifies the file to receive the load map. If you omit this parameter, the load map is written to file $LOCAL.LOADMAP. The LOAD_MAP_OPTIONS parameter specifies the particular load map options. The options you can select are:

| | |
|---|---|
| S | Segment map |
| CR | Entry points cross reference map |
| B | Block map |
| ALL | Selects S, B, EP, and CR options |
| EP | Entry points map |
| NONE | Suppresses load map generation |

For example, the following command loads and executes a program and produces a load map:

```
/execute_task file=lgo load_map=mapfile load_map_options=(b,ep,cr)
```

This command produces a block, entry points, and entry points cross reference map. The map is written to file MAPFILE.

The EXECUTE_TASK command parameters described above select load map options for a single load operation. You can use the SET_PROGRAM_ATTRIBUTES (SETPA) command to set load map options for the duration of the job. The parameters for controlling load map options have the same form as for the EXECUTE_TASK command:

```
SET_PROGRAM_ATTRIBUTE LOAD_MAP=file LOAD_MAP_OPTIONS=(option_list)
```

The LOAD_MAP parameter specifies the file to receive the load map, and the LOAD_MAP_OPTIONS specifies the contents of the map. Options can be any combination of the symbols EP (entry points map), B (block map), CR (entry points cross reference map), or S (segment map). You can suppress load map generation by specifying LOAD_MAP_OPTIONS=NONE.

The SET_PROGRAM_ATTRIBUTE command changes the job default load map options. The new default remains in effect until you either change it with another SET_PROGRAM_ATTRIBUTES command or override it for a particular load operation with an EXET command.

The following example illustrates the use of SET_PROGRAM_ATTRIBUTES to set load map options:

```
/set_program_attributes load_map=mfile load_map_options=(b,ep,cr)
/execute_task file=(aa,bb)
/execute_task file=lgo
```

The SET_PROGRAM_ATTRIBUTES command requests a block, entry points, and entry points cross reference map, and specifies that file MFILE is to receive the map. The requested maps are produced by the two subsequent EXECUTE_TASK operations.

The following example shows how an EXECUTE_TASK command can override a SET_PROGRAM_ATTRIBUTES command for a particular execution:

```
/set_program_attributes load_map_options=b
/execute_task file=lgo
/execute_task file=(cc,dd) load_map_options=none
```

The SET_PROGRAM_ATTRIBUTES command requests a block map. The first EXECUTE_TASK command produces this map. Since no LOAD_MAP parameter is specified, the map is written to file $LOCAL.LOADMAP. The LOAD_MAP_OPTIONS=NONE parameter on the second EXECUTE_TASK command turns the load map options off for that execution.

## Presetting Memory

Under both NOS and NOS/VE, you can specify a memory preset value at load time. Under NOS/VE, the initial default preset value is zero. You can change this default for a job through the SET_PROGRAM_ATTRIBUTES command. You can change the preset value for a particular execution by specifying the PRESET_VALUE parameter on an EXECUTE_TASK command. The PRESET_VALUE parameter has the form

```
EXECUTE_TASK FILE=files PRESET_VALUE=value
```

The specified value is one of the following:

| | |
|---|---|
| ZERO | Presets memory to zero. |
| FLOATING_POINT_INDEFINITE | Presets memory to floating point indefinite. |
| INFINITY | Presets memory to floating point infinity. |
| ALTERNATE_ONES | Presets memory to an alternating bit pattern with the high-order bit set to one. |

For ease of debugging, you should preset memory to FLOATING_POINT_INDEFINITE or INFINITY. This ensures that your program will terminate when uninitialized variables are used in computations. A preset value of zero, which is a legitimate value, would allow the program to continue executing.

For example, the following command presets memory to infinity and begins execution of the program on file LGO:

```
/execute_task  file=lgo  preset_value=infinity
```

## Debugging Options

The EXECUTE_TASK command provides several debugging options intended for use with the Debug utility.

The Debug utility enables you to debug your program during execution. It is similar to the CYBER Interactive Debug facility (CID). Through Debug, you can suspend execution of your program at selected points or when specified conditions occur. While execution is suspended, you can display or change the values of variables and arrays in the program. You can then resume execution of the program. Debug provides the following advantages over conventional debugging techniques:

You can examine program values at the precise time of an error or at other desired times.

It requires no changes to your source program.

You need to know only a few commands.

Refer to the Debug Usage manual for more information on Debug.

The EXECUTE_TASK command provides the following parameters for controlling the use of the Debug facility for an execution:

| | |
|---|---|
| DEBUG_MODE | Turns debug mode on or off for the execution. When debug mode is on, the execution takes place under Debug control. Options are DEBUG_MODE=ON and DEBUG_MODE=OFF. |
| DEBUG_INPUT | Specifies the file from which Debug reads commands. Default is $INPUT (commands read from the terminal). |
| DEBUG_OUTPUT | Specifies the file to which Debug writes output. Default is $OUTPUT (output displayed at terminal). |
| ABORT_FILE | Specifies a file of Debug commands that Debug will execute if an execution error occurs. (DEBUG_MODE must be set to OFF.) |

## Summary of NOS and NOS/VE Loader Differences

Specify loader options:

NOS: Load Sequences.

NOS/VE: Use EXECUTE_TASK (EXET) command to specify loader and execution-time options.

Specify a name call load:

NOS: Specify object file name (default is LGO).

NOS/VE: Specify object file name (default is LGO).

Load modules from multiple files:

NOS: LOAD statement.

NOS/VE: FILE parameter on EXECUTE_TASK command.

Load modules from libraries:

NOS: LIBLOAD statement.

NOS/VE: MODULES parameter on EXECUTE_TASK command.

Begin execution:

NOS: EXECUTE or name call statement.

NOS/VE: EXECUTE_TASK or name call command.

Request load map:

NOS: LDSET(MAP=...).

MAP statement.

NOS/VE: LOAD_MAP and LOAD_MAP_OPTIONS parameters on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES (SETPA) command.

Specify libraries to be searched:

NOS: LDSET(LIB=...) or LIBRARY statement.

NOS/VE: LIBRARY parameter on EXECUTE_TASK command or ADD_LIBRARY parameter on SET_PROGRAM_ATTRIBUTES command.

Preset memory:

NOS: LDSET(PRESET=...) and LDSET(PRESETA=...) statements.

NOS/VE: PRESET_VALUE parameter on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command.

Load selected programs from a library:

NOS: LIBLOAD statement.

NOS/VE: MODULES parameter on EXECUTE_TASK command.

Specify error termination conditions:

NOS: LDSET(ERR=...) statement.

NOS/VE: TERMINATION_ERROR_LEVEL parameter on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command.

Interactive debugging options:

NOS: DEBUG,ON statement.

DEBUG,OFF statement.

Post Mortem Dump.

NOS/VE: DEBUG_MODE=ON parameter on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command.

DEBUG_MODE=OFF parameter on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command.

ABORT_FILE parameter on EXECUTE_TASK or SET_PROGRAM_ATTRIBUTES command specifies a file of Debug commands to be automatically executed if an execution error occurs.

Satisfy External References:

NOS: SATISFY statement.

name call statement.

EXECUTE statement.

NOS/VE: name call command.

EXECUTE_TASK command.

## Invoking the APL System

Under NOS/VE, you enter the APL system by using the APL command, described by the following topics:

Comparison of NOS APL and NOS/VE APL Commands
NOS/VE APL Command Parameters

### Comparison of NOS APL and NOS/VE APL Commands

The command to invoke APL under NOS/VE is:

```
APL option option option ...
```

The APL command follows the same syntax rules as other SCL commands. Parameters have the form

```
keyword=value
```

Parameters are separated by a comma or a space.

The following table presents a quick comparison of the NOS and NOS/VE APL command parameters. The table shows the NOS parameter name, the NOS/VE parameter name, the NOS/VE parameter abbreviation in parentheses, and a short description of each parameter.

In some cases, the possible values that can be specified for a parameter differ between NOS and NOS/VE. For a summary of all of the NOS/VE APL parameters and their possible values, see the APL Language Definition Usage manual.

| NOS | NOS/VE | Description |
|---|---|---|
| I | INPUT (I) | File containing APL statements. |
| L | OUTPUT (O) | File to receive output. |
| LO | LIST_OPTIONS (LO) | Output listing options. |
| LO=E | LO=S | - Echo (copy) input to output. |
| LO=P | LO=P | - Prohibit prompt. (Under NOS/VE, prohibits 6 space prompt and line number prompt in function definition mode.) |
| LO=B | — | - Under NOS, placed a blank in first column of output lines; LO=B has different meaning under NOS/VE. |
| NH parameter | LO=B | - Suppresses APL system banner from being sent to output file. Under NOS, the NH parameter provided this function; under NOS/VE, LO=B provides this function. |
| NH | (see LO parameter) | |
| — | STATUS | Specifies an SCL status variable to receive an error status code. |
| TT | TERMINAL_TYPE (TT) | Specifies the type of terminal you are using. |
| TT=ASCAPL | TT=APL | - APL terminal using full APL set. |
| TT=COR | TT=COR | - Correspondence Code Selectric terminal. |
| TT=TYPE | — | - Typewriter-paired APL terminal; NOS/VE network provides translation for these. |
| TT=BIT | — | - Bit-paired APL terminal; NOS/VE network provides translation for these. |
| TT=TTY33 | — | - Teletype 33 terminal. |
| TT=ASCII | TT=ASCII | - Full ASCII terminal. |
| TT=BATCH | TT=BATCH | - 64 ASCII character set printer. |
| TT=B501 | — | - Batch 501 printer. |
| TT=TTY383 | — | - Some kind of teletype 38 terminal. |
| TT=713 | TT=UCA | - Full ASCII terminals without APL character set. |
| WS | WORKSPACE (WS) | Specifies the workspace to be automatically loaded. |

| NOS | NOS/VE | Description |
|---|---|---|
| PW | PASSWORD (PW) | Specifies the password for the file specified by WS. |
| UN | (not needed) | |
| MX | (not needed) | |
| MN | (not needed) | |
| -- | WAIT (W) | Causes APL to wait for the file specified by WS to become available if that file is busy. |

## NOS/VE APL Command Parameters

The parameters of the NOS/VE APL command are briefly summarized on the following screens. The information about each parameter includes the full parameter name, the abbreviation in parentheses, default values, and other allowed values. The parameters are:

INPUT

LIST_OPTIONS

OUTPUT

PASSWORD

STATUS

TERMINAL_TYPE

WAIT

WORKSPACE

### INPUT (I)

| | |
|---|---|
| default | $INPUT (For interactive use, $INPUT is the terminal.) |
| I=file | (For batch use, I=file must specify the file containing the APL statements to be interpreted.) |

### LIST_OPTIONS (LO)

You can omit this parameter entirely or specify it with any combination of the values B, S, and P below.

| | |
|---|---|
| default | No options set (For interactive use)<br>LO=SP (For batch use) |
| LO=B | Suppresses sending of APL banner to the output file. |
| LO=S | Copies all input to the output file. |
| LO=P | Suppresses sending of prompts to the output file; prompts suppressed are the standard 6 spaces and the line number prompt in function definition mode. |

**OUTPUT (O)**

| | |
|---|---|
| default | $OUTPUT (For interactive use, $OUTPUT is the terminal; for batch use, $OUTPUT is the file normally printed at the end of the job.) |
| O=file | Specifies the file to which your output will be sent. |

**PASSWORD (PW)**

| | |
|---|---|
| default | No password. |
| PW=password | Specifies the password required to access the file specified in the WS parameter. |

**STATUS**

| | |
|---|---|
| default | Displays error status in the output file. |
| STATUS=status-var | Specifies an SCL status variable to receive the status of the APL job step when the job step completes. |

**TERMINAL_TYPE (TT)**

| | |
|---|---|
| default | TT=APL |
| TT=APL | Specifies that your terminal is an APL terminal using the full APL character set. |
| TT=COR | Specifies that your terminal is an APL terminal using the APL character set but lacking several APL characters. (See the APL Language Definition Usage manual for the list of characters.) Specify this option if you are using an IBM 2741 Correspondence Code Selectric terminal with an APL type ball. |
| TT=UCA | Specifies that your terminal is an ASCII-compatible terminal without the APL character set. This option causes lowercase letters to be translated to upper case ASCII letters on input. This option uses the $ escape sequences for input and output of characters not in the ASCII character set. (See the APL Language Definition Usage manual for escape sequences.) |
| TT=ASCII | Specifies that your terminal is an ASCII-compatible terminal without the APL character set. This option uses the $ escape sequences for input and output of characters not in the ASCII character set. (See the APL Language Definition Usage manual for the $ escape sequences.) |
| TT=BATCH | Specifies that input and output use only the Control Data 64 ASCII character set graphics. Use this option to print output on CDC batch printers that cannot display all 95 ASCII characters. This option uses the $ escape sequences for input and output of characters not in the ASCII character set. (See the APL Language Definition Usage manual for the $ escape sequences.) |

**WAIT (W)**

| | |
|---|---|
| default | Same as W=FALSE. |
| W=TRUE | Causes the APL system to wait for the file specified by WS to become available if that file is busy. |
| W=FALSE | Causes the APL system to terminate with an abnormal status if the file specified by WS is busy. |

**WORKSPACE (WS)**

| | |
|---|---|
| default | :$SYSTEM.APL.CLEARWS (the clear workspace) |
| WS=file | Specifies the file to be loaded automatically when the APL system begins execution. |

# Using Object Libraries 8

# Using Object Libraries 8

Both NOS and NOS/VE enable you to create specially-formatted files containing object modules. Under NOS, these files are called user libraries; under NOS/VE, they are called object libraries. (From now on, only the term object libraries will be used.)

However, under both systems, libraries have the same purpose: They provide an efficient method of maintaining files of object programs, and the loader can quickly locate object modules in libraries while satisfying external references. (Satisfying external references is discussed later in this chapter.)

Under NOS, you use the LIBGEN utility to create user libraries. Under NOS/VE, you use a utility called CREATE_OBJECT_LIBRARY to create object libraries. These utilities are similar in that they accept as input a file containing compiled object modules (such as the binary output file produced by FORTRAN or COBOL), and produce as output an object library containing the object modules.

Under both systems, you can maintain an object library by adding, deleting, and replacing modules in the library. However, under NOS, you cannot perform these update operations directly to the library. It is necessary to use LIBEDIT to perform the additions, deletions, and replacements to the original object file that you used to create the library. You then input the updated object file to LIBGEN to create a new object library.

Under NOS/VE, you can use CREATE_OBJECT_LIBRARY to add, delete, and replace modules in the object library itself (although CREATE_OBJECT_LIBRARY actually creates a new object library with changes, rather than updating an existing library).

NOS/VE object libraries can contain other types of modules besides object modules. You can place SCL procedures on libraries, so that you can execute the procedure by referencing the procedure name. You can also place a special type of a module called a program description on an object library. (A program description module provides an alternative to the EXECUTE_TASK command for executing a program. Refer to the SCL Object Code Management Usage manual for more information about program description modules.)

The following examples illustrate the creation of an object library under NOS and NOS/VE. Modules from the object file LGO are placed in the library.

```
NOS:     LIBGEN,F=LGO,P=LIBFIL,N=MYLIB

NOS/VE:  create_object_library
         add_modules library=lgo
         generate_library library=mylib
         quit
```

Under NOS/VE, you create object libraries during a CREATE_OBJECT_LIBRARY session. During the session, you enter commands that specify the modules to be placed in the library. Although, under NOS/VE, more work is required for simple operations such as the one shown above, commands are provided that enable you to perform many additional operations on object libraries.

## Using CREATE_OBJECT_LIBRARY

Under NOS/VE, you create object libraries by using the CREATE_OBJECT_LIBRARY utility. CREATE_OBJECT_LIBRARY is similar to the NOS LIBGEN/LIBEDIT utilities. The libraries can contain modules from binary object files, procedures, or existing libraries. Thus, you can use CREATE_OBJECT_LIBRARY to create updated versions of existing libraries.

The library creation and modification operations are performed during a CREATE_OBJECT_LIBRARY session. A CREATE_OBJECT_LIBRARY session consists of the following steps:

1. Begin the session.
2. Enter commands to add, delete, or replace modules in the library.
3. Generate the library.
4. End the session.

The command to begin a CREATE_OBJECT_LIBRARY session is:

```
CREATE_OBJECT_LIBRARY  (Abbreviated CREOL)
```

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 RETURN, 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 session and returns control to the operating system.

While you are in a CREATE_OBJECT_LIBRARY session you enter commands that provide CREATE_OBJECT_LIBRARY with the names of files that contain modules to be added to the library, modules to be deleted, modules to be replaced, and replacement modules.

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, delete, and replace modules on this list. To actually create a new library (or a modified version of an existing one), you must enter the GENERATE_LIBRARY (GENL) command:

```
GENERATE_LIBRARY FILE=file_name  (Abbreviated GENL F=file_name)
```

This command, entered during a CREATE_OBJECT_LIBRARY session, creates an object library that contains the modules in the module list, and writes the library to the specified file. Typically, the GENERATE_LIBRARY command is the last command you enter before ending a CREATE_OBJECT_LIBRARY session.

After you have generated the library, you can end the session by typing the command

```
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 CREATE_OBJECT_LIBRARY 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 (ADDM) command. This command has the form:

```
ADD_MODULES LIBRARY=file_list  (Abbreviated ADDM L=file_list)
```

where file_list specifies one or more files that contain object modules to be added to the library.

The files you specify can be object files produced by a compiler, or they can be other libraries. CREATE_OBJECT_LIBRARY adds all the modules from the specified files to the object library.

The ADD_MODULES command adds all the modules from the specified files to the object library. You can add selected modules from one or more files by including the MODULE parameter on the ADD_MODULES command:

```
ADD_MODULES LIBRARY=file_list MODULE=mod_list
```

This command directs CREATE_OBJECT_LIBRARY to add to the library only the modules listed in mod_list. CREATE_OBJECT_LIBRARY searches for the listed modules on the files specified by the LIBRARY parameter.

For example, the following command, entered during a CREATE_OBJECT_LIBRARY session, adds three modules to the internal module list:

```
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 the module list, you must enter a GENERATE_LIBRARY command to actually create the object library.

## Modifying Object Libraries

CREATE_OBJECT_LIBRARY enables you to add, replace, and delete modules from existing object libraries. The following list summarizes the commands you use to perform these modifications:

| | |
|---|---|
| ADD_MODULES | Adds one or more modules to an object library. |
| REPLACE_MODULES | Replaces one or more modules in an object library. |
| DELETE_MODULES | Deletes one or modules from an object library. |

The ADD_MODULES command can be used to add modules to an existing library or to a new library.

These commands can be entered only during a CREATE_OBJECT_LIBRARY session (that is, after you have entered a CREATE_OBJECT_LIBRARY command).

Remember that the CREATE_OBJECT_LIBRARY commands do not directly alter the contents of an object library. Instead, the commands alter the contents of the module list. Therefore, when you begin a 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_name
```

The library_name is the library you want to modify. You can then use the ADD_MODULES, REPLACE_MODULES, and DELETE_MODULES parameters to alter the library list.

Then, after you make all the desired changes to the module list, you use the GENERATE_LIBRARY command to rewrite the existing library.

The REPLACE_MODULES (REPM) command replaces modules in the module list. This command has the form

```
REPLACE_MODULES FILES=file_list MODULES=mod_list  (Abbreviated REPM F=file_list M=mod_list)
```

The file_list specifies one or more files containing replacement modules, and the 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 module list are eligible for replacement.

When CREATE_OBJECT_LIBRARY processes 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 seaching the entire module list, the replacement module is disregarded (it is not added to the module list) and processing continues with the next module in the file (or in the mod_list, if you specified the MODULES parameter).

Processing continues in this manner until the file-list (or the mod-list, if specified) is exhausted.

The following REPLACE_MODULES command, entered during a CREATE_OBJECT_LIBRARY session, specifies three modules to be replaced:

```
COL/replace_modules file=bin_file module=(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.

The DELETE_MODULES (DELM) command deletes modules from the module list. This command has the form

```
DELETE_MODULES MODULES=mod_list  (Abbreviated DELM M=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 module list.

After you have specified all the modules to be added to, deleted from, or replaced in the module list, you create a library that contains the modules currently in the module list by entering the GENERATE_LIBRARY (GENL) command:

```
GENERATE_LIBRARY LIBRARY=library_name  (Abbreviated GENL L=library_name)
```

The library you specify can be either an existing library or a new one. If you specify an existing library, it is replaced. For example:

```
create_object_library
add_modules library=lgo
generate_library library=proglib
quit
```

These commands begin a CREATE_OBJECT_LIBRARY session, add modules from file LGO to the (initially empty) module list, and generate a library named PROGLIB that contains the modules in the module list. If PROGLIB already exists (as a local file), it is replaced.

## Displaying Information About Object Libraries

You can display a list of the modules in a library or object file by entering a DISPLAY_OBJECT_LIBRARY (DISOL) command. This command is similar to the NOS ITEMIZE command, which displays information about the records in binary files under NOS. The simplest form of the DISPLAY_OBJECT_LIBRARY command is:

```
DISPLAY_OBJECT_LIBRARY LIBRARY=file_name  (Abbreviated DISOL L=file_name)
```

where file-name is the object file name or library file name. The DISOL 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 file. You can request more information through additional parameters, as described in the Object Code Management manual.

The DISOL command is an SCL command, rather than a CREATE_OBJECT_LIBRARY command. That means that you can enter it either within or outside of a CREATE_OBJECT_LIBRARY session. You can use DISOL to display information about the modules in the library you are currently updating or about any other binary file under NOS/VE.

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
```

Now, suppose the following CREATE_OBJECT_LIBRARY session is executed:

```
/create_object_library
COL/add_modules files=(bin_file_1,bin_file_2,) modules=(a,b,e,f)
COL/generate_library file=mylib
COL/quit
/display_object_library library=mylib
```

This session creates an object library that contains four modules. The DISPLAY_OBJECT_LIBRARY command displays the following 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
```

## Summary of Using Object Libraries

The following is a summary of the NOS commands and corresponding NOS/VE commands for creating and modifying object libraries.

Create a library:

NOS: LIBGEN.

NOS/VE: CREATE_OBJECT_LIBRARY to begin the session.

ADD_MODULES to add modules to the library.

GENERATE_LIBRARY to create the library.

QUIT to end the session.

Modify a library:

NOS: LIBEDIT modifies the sequential binary object file.

LIBGEN creates a new library.

NOS/VE: CREATE_OBJECT_LIBRARY begins the session.

ADD_MODULES adds modules to the module list.

DELETE_MODULES deletes modules from the module list.

REPLACE_MODULES replaces modules in the module list.

GENERATE_LIBRARY creates a new library.

QUIT terminates the session.

Display Information about a library:

NOS: ITEMIZE

NOS/VE: DISPLAY_OBJECT_LIBRARY

# Submitting Batch Jobs 9

# Submitting Batch Jobs 9

You can submit batch jobs under both NOS and NOS/VE. Under both systems, a batch job is a file of commands and data that the system processes as a unit. The formats of batch jobs, and the commands for submitting batch jobs, differ under NOS and NOS/VE.

## Batch Job Format

Under NOS and NOS/VE, a batch job is simply a coded file that contains the commands to be executed and any data required by the commands. Under NOS/VE, the first line of a batch job must be a LOGIN command. This command provides the system with the information necessary to validate the job. The LOGIN command has the form:

```
LOGIN USER=user_name PASSWORD=password FAMILY_NAME=family_name
```

Typically, you use the same validation information you used to log in at the terminal.

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

The following NOS and NOS/VE examples compile and execute a FORTRAN program. The source program is contained in a file named FTNSRCE.

```
NOS:     MYJOB.
         USER, MYNAME, MYPASS.
         CHARGE, 12345,6789.
         ATTACH,FTNSRCE.
         FTN5,I=FTNSRCE,L=FLIST.
         LGO.
         SAVE,FLIST.

NOS/VE:  login user=myname password=mypass family_name=myfam
         attach_file file=$user.ftnsrce
         create_file file=$user.flist
         fortran input=ftnsrce list=flist
         lgo
         logout
```

## Creating a Batch Job

You can create a file containing a batch job using a text editor such as the Full Screen Editor. Alternatively, you can use the COLLECT_TEXT command. This command provides a quick way of creating a text file. The COLLECT_TEXT command has the form:

```
COLLECT_TEXT OUTPUT=file_name  (Abbreviated COLT O=file_name)
```

where file_name is the file to contain the text. 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 ** in columns 1 and 2. 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, the following COLLECT_TEXT command creates a file containing a batch job:

```
/collect_text output=batch_comp  <---------------------- Begin text mode.
ct? login user=urpass password=urpass family_name=urfam
ct? cobol input=$user.cobsrce binary=cobbin list=$list
ct? cobbin
**  <---------------------------------------------------- End text mode.
```

File BATCH_COMP contains a LOGIN command, a COBOL command, and a name call command.

Under both systems you can create batch jobs that contain input data for commands in the job. Under NOS, input data follows the command record and is separated from the command record by an end-of-record indicator.

NOS/VE files do not have boundaries corresponding to the NOS record boundaries. However, you can create a batch job that contains input data by including a COLLECT_TEXT command in the job. The COLLECT_TEXT command creates a local file that contains the data. You then input that file to the command that requires the data.

The following NOS batch job compiles and executes a FORTRAN program. The FORTRAN source and the execution-time input data are both read from file INPUT.

```
MYJOB.
USER,MYNAME,MYPASS.
CHARGE,12345.
FTN5.
LGO.
--EOR--
   FORTRAN source program
--EOR--
   Input data for FORTRAN program
--EOI--
```

The following NOS/VE batch example compiles and executes a FORTRAN program. The source program and the execution-time input data are both contained in the batch job. COLLECT_TEXT commands are used to create a local file named SRCE, which contains the FORTRAN source, and a local file named INDAT, which contains the input data.

```
login user=myname password=mypass family=myfam
collect_text output=srce  <------------------- Create file SRCE containing source program.
    .
    .
    .
    FORTRAN source program
    .
    .
    .
**
fortran input=srce  <------------------------- FORTRAN compiler reads file.
collect_text output=indat  <------------------ Create file INDAT containing input data.
    .
    .
    .
    Input data for FORTRAN program
    .
    .
    .
**
lgo
```

## Command to Submit a Batch Job

The NOS/VE command to submit a batch job is SUBMIT_JOB (SUBJ). This command is similar to the NOS SUBMIT and ROUTE commands. The SUBMIT_JOB command has the form:

```
SUBMIT_JOB FILE=file_name JOB_NAME=job_name  (Abbreviated SUBJ F=file_name JN=job_name)
```

The file_name is the file containing the batch job, and job_name is an arbitrary name that you assign to the job. You can use the job name to reference the job, in much the same way you use the system-assigned job sequence name (jsn) under NOS. (NOS/VE also assigns a unique name that it uses to identify the job. You can reference the job by either name.)

Following are equivalent NOS and NOS/VE examples of submitting batch jobs:

```
NOS:     SUBMIT, JOBFIL.

NOS/VE:  submit_job file=jobfil job_name=myjob
```

Both commands submit a batch job contained in file JOBFIL. The NOS/VE example assigns the name MYJOB to the job (overriding any JOB_NAME parameter on the LOGIN command). In the NOS example, the job name is specified on the job statement.

## Displaying Job Status Information

After you have submitted a batch job under NOS/VE, you can check the progress of the job with the DISPLAY_JOB_STATUS (DISJS) command. This command provides status information similar to that provided by the NOS ENQUIRE command. The DISPLAY_JOB_STATUS command has the form

```
DISPLAY_JOB_STATUS JOB_NAME=job_name  (Abbreviated DISJS JN=job_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 state 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.

The following example shows typical output for a DISPLAY_JOB_STATUS command:

```
Command:

    /display_job_status job_name=myjob

Output:

    System_Supplied Name       : aaq$
    User_Supplied Name         : myjob
    Originating_User           : myname
    Originating_Family         : myfam
    Job_Class                  : batch
    Job_Mode                   : batch
    Job_State                  : execute
    Operator_Action_Posted     : no
    Display_Message            : attf file=progfile <-- This command was executing.
```

The Display_Message entry shows the command that was executing when the DISPLAY_JOB_STATUS command was entered.

The NOS/VE job log is similar to the NOS dayfile: It contains a chronological history of the operations that occurred during the job. The job log for a batch job is always printed at the end of the job. You can preserve a copy of the job log for later examination by including a DISPLAY_LOG command in the job. This command has the form:

```
DISPLAY_LOG OUTPUT=permanent_file  (Abbreviated DISL O=permanent_file)
```

where permanent_file has the form $USER.filename. This command writes the job log to the specified permanent file. After the batch job has completed, you can examine the job log by displaying the permanent file at your terminal.

By including a WHEN/WHENEND block in a batch job, you can specify commands to be executed if an error occurs during job processing. The function of the WHEN/WHENEND block is similar to that of the NOS EXIT statement. The format of the WHEN/WHENEND block is:

```
WHEN ANY_FAULT DO
   commands
WHENEND
```

If any system or user program terminates with an error during job processing, the commands between WHEN and WHENEND are executed. If no error occurs, the commands are ignored. (Refer to the SCL Language Definition Usage manual for more information on error condition processing.)

The following batch job contains a WHEN statement that is executed if an error occurs during execution of a command or program:

```
login user=myname password=mypass family=myfam
when program_fault do
   put_line line=' '//'error is: '//$strrep(osv$status)
whenend
attach_file file=$user.newdata
create_file file=$user.flist
fortran input=$user.source list=flist
lgo
```

If an error occurs during execution of any of the commands in the job, the contents of the system status variable OSV$STATUS are printed. The job then terminates. If no errors occur, the job runs to completion and the status variable is not printed.

## Summary of Submitting Batch Jobs

The following summary shows corresponding NOS and NOS/VE commands for submitting batch jobs.

Batch job format:

```
NOS:     Job statement
         USER statement
         CHARGE statement
         .
         .
         .
         Control statements
         .
         .
         .
         end-of-information
```

```
NOS/VE:  LOGIN
         .
         .
         .
         commands
         .
         .
         .
```

Input data in a batch job:

```
NOS:     Job statement
         USER statement
         CHARGE statement
         .
         .
         .
         Control statements
         .
         .
         .
         end-of-record
         .
         .
         .
         Data
         .
         .
         .
         end-of-information

NOS/VE:  LOGIN command
         COLLECT_TEXT OUTPUT=name
         .
         .
         .
         Data
         .
         .
         .
         **
         .
         .
         .
         Commands (read data from file created by COLLECT_TEXT).
         .
         .
         .
```

Submit a batch job:

```
NOS:     SUBMIT,file.
         ROUTE,file,DC=IN.

NOS/VE:  SUBMIT_JOB FILE=file JOB_NAME=name
```

Display status information about a batch job:

NOS: ENQUIRE,jsn

NOS/VE: DISPLAY_JOB_STATUS JOB_NAME=name

Transfer control on error:

NOS: EXIT command causes transfer of control if error occurs.

NOS/VE: WHEN/WHENEND block specifies commands to be executed if error occurs.

# Migrating Files

# File Interface Introduction 10

# File Interface Introduction 10

This discussion describes the file organizations available on NOS/VE and briefly compares them to similar organizations on NOS. The discussion tells how files are described to the system and the kinds of files typically used by FORTRAN and COBOL programs. The file organizations listed below are available on NOS/VE; the actual key (AK) file organization is not available on NOS/VE.

Sequential
Byte addressable
Indexed sequential
Direct Access

Other information about files is also discussed in this chapter as follows:

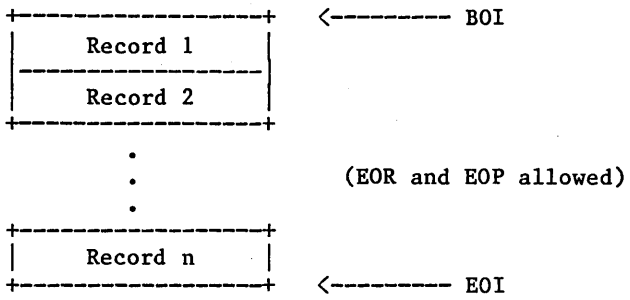
NOS/VE Record types
File attributes
File attributes used by FORTRAN programs
File attributes used by COBOL programs

## Sequential File Organization

Records in a sequential file are stored and retrieved in the order the records are presented to the system. The records are logically contiguous.

The records can be accessed sequentially.

The logical structure of a sequential file is as follows:




Sequential files have the following characteristics:

- They can be assigned to the following devices: mass storage, terminal, magnetic tape, and null.
- All three record types (CDC-variable, ANSI-fixed, and undefined) are available.
- Both types of blocking (system-specified and user-specified) can be used.
- They can be positioned forward (toward EOI) or backward (toward BOI) by a specified number of records or partitions.
- Partial records can be read or written. Partitioning is available with CDC-variable record type.

NOS and NOS/VE sequential files are compared in table 10-1.

Table 10-1. Comparison of NOS and NOS/VE Sequential Files

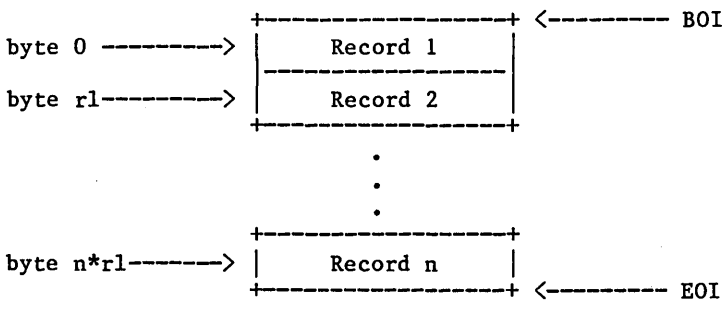
| NOS | NOS/VE |
|---|---|
| Records stored in order presented | Records stored in order presented |
| Records logically contiguous | Records logically contiguous |
| Records accessed sequentially | Records accessed sequentially |
| Files can be assigned to mass storage, terminal, or magnetic tape | Files can be assigned to mass storage, terminal, magnetic tape, or null |
| 4 block types available (I, C, K, E) | 2 block types available (system-specified and user-specified) |
| 8 record types available (D, F, R, S, T, U, W, Z) | 3 record types available (CDC-variable, ANSI-fixed, and undefined) |

## Byte Addressable File Organization

Records in a byte addressable file are stored and retrieved according to the byte address at the start of the record. Bytes in the file are numbered sequentially from zero; that is, 0, 1, 2, and so forth.

Records can be accessed randomly and sequentially.

The logical structure of a byte addressable file is as follows when the length of the record is rl bytes and n represents the number of records:




A byte addressable file can be assigned to the following devices: mass storage, terminal, and null.

All three record types (CDC-variable, ANSI-fixed, and undefined) can be used with byte addressable files.

Both block types (system-specified and user-specified) can be used with byte addressable files.

The file cannot be positioned forward or backward by records or partitions.

Partial records can be read to or written from a byte addressable file. Partitioning is available with CDC-variable record type.

The NOS/VE byte addressable file organization is similiar to the NOS word addressable file organization. The major characteristics of the two file organizations are compared in table 10-2.

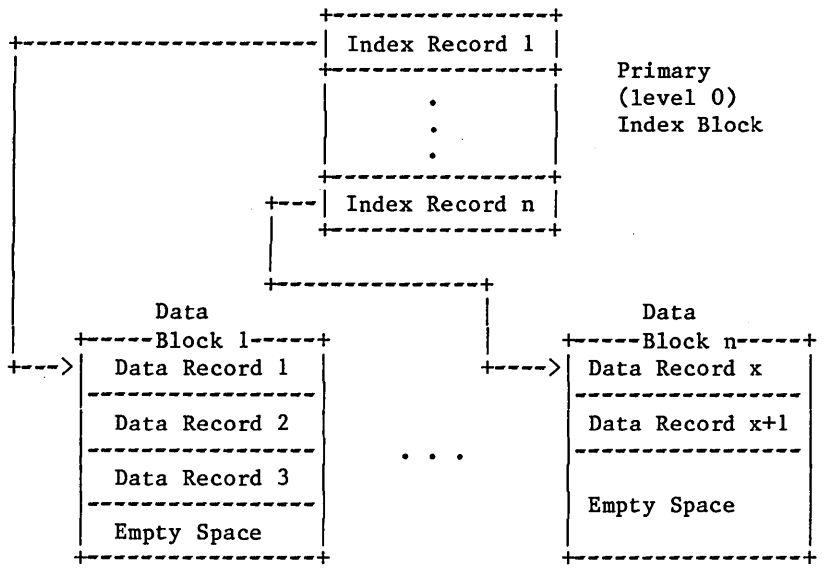
Table 10-2. Comparison of Word Addressable and Byte Addressable Files

| Word Addressable | Byte Addressable |
|---|---|
| Record access by word address | Record access by byte address |
| Random and sequential access | Random and sequential access |
| Words numbered from 1 to n | Bytes numbered from 0 to n |
| 10 characters per word | 1 character per byte |
| Records can be stored on mass storage | Records can be assigned to mass storage, terminal, or null |
| Record types W, F, and U allowed | Record type CDC-variable, ANSI-fixed, and undefined allowed |
| No logical boundaries between records | Partitioning allowed with CDC-variable record type |

## Indexed Sequential File Organization

Records in an indexed sequential file are stored in ascending order according to primary key value. The primary key is a group of contiguous bytes that uniquely identify the record.

Records in an indexed sequential file can be accessed randomly by primary key value or sequentially by the logical position of the record. The logical structure of an indexed sequential file is shown in the following diagram.



On NOS/VE, indexed sequential files can be nested; that is, an indexed sequential file can be defined within a NOS/VE file cycle. Currently, the only way to create a nested file is through a CYBIL program.

The NOS and NOS/VE indexed sequential files are similar. The major characteristics of the file organization is compared in table 10-3.

Table 10-3. Comparison of NOS and NOS/VE Indexed Sequential Files

| NOS | NOS/VE |
|---|---|
| Records stored in order according to primary key values | Records stored in order according to primary key value |
| Records accessed sequentially by position or randomly by key | Records accessed sequentially by position or randomly by key |
| 8 record types available (D, F, R, S, T, U, W, Z) | 3 record types available (CDC-variable, ANSI-fixed, undefined) |
| 15 levels of indexing allowed | 15 levels of indexing allowed |
| Alternate keys supported by FORTRAN and COBOL | Alternate keys supported by FORTRAN and COBOL |
| Cannot be nested | Can be nested |

## Direct Access File Organization

The direct access file organization is like the indexed sequential file organization in its use of a primary key. Like an indexed sequential file, a direct access file can have alternate keys.

As with indexed sequential files, you must specify the primary key value when writing or deleting a direct access file record. Similarly, you must specify either a primary key value or an alternate key value to read a direct access file record.

Direct access files differ from indexed sequential files in their ordering of records in the file. When you read records sequentially from an indexed sequential file, records are returned in order sorted by their primary key value. Direct access records can be read sequentially, but their primary key values are in random order.

Direct access files are divided into blocks called home blocks. When a record is written to a direct access file, its primary key value is hashed by an internal hashing procedure to generate the number of the home block in which the record is written. The hashing procedure distributes the records uniformly among the home blocks.

On NOS/VE, direct access files can be nested; that is, a direct access file can be defined within a NOS/VE file cycle. Currently, the only way to create a nested file is through a CYBIL program.

The following diagram shows how records are written to a direct access file. A record having primary key XYZ is written to block 3 of a direct access file.

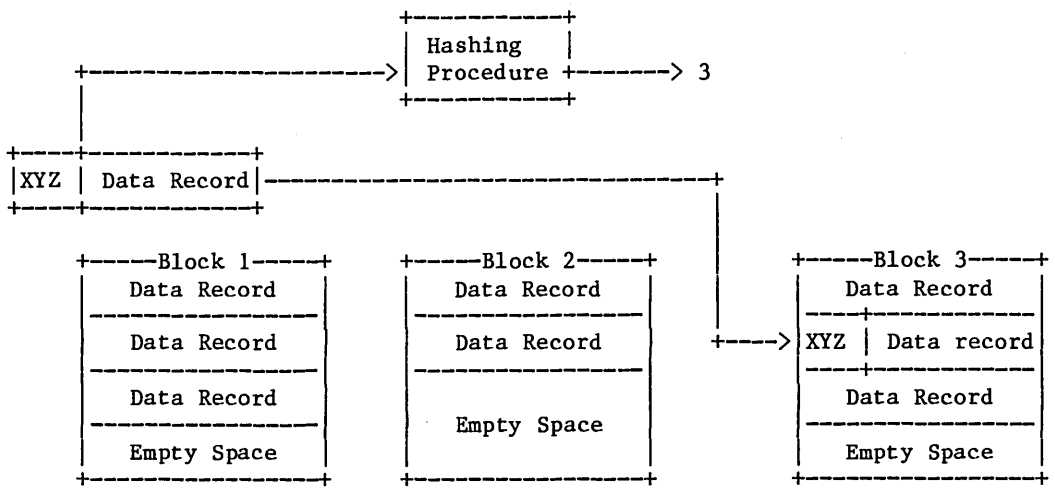



Table 10-4 compares direct access file characteristics under NOS and NOS/VE.

Table 10-4. Comparison of NOS and NOS/VE Direct Access File Characteristics.

| NOS | NOS/VE |
|---|---|
| Records stored in blocks according to hashed value of primary key. | Records stored in blocks according to hashed value of primary key. |
| Records accessed sequentially by position or randomly by key. | Records accessed sequentially by position or randomly by key. |
| 8 record types available (D, F, R, S, T, U, W, Z). | 3 record types available (CDC-variable, ANSI-fixed, undefined). |
| Supported by FORTRAN and COBOL. | Supported by COBOL, but not by FORTRAN. |
| Alternate keys supported. | Alternate keys supported. |
| Cannot be nested. | Can be nested. |

## NOS/VE Record Types

Three types of records are available on NOS/VE. The NOS/VE record types and NOS equivalents are as follows:

CDC-variable length; similar to word count (W) on NOS

ANSI-fixed length; similar to fixed length (F) on NOS

Undefined; similar to undefined (U) on NOS

NOS record types decimal character count (D), record mark (R), system (S), trailer count (T), and zero byte (Z) are not available on NOS/VE.

### CDC-Variable Record Type

The variable record type is the default record type. Variable records can be any length except for files of indexed sequential or direct access organization where a minimum record length applies as defined for the file.

The variable record type supports fixed or variable record lengths, partial record I/O, and file partitioning.

Each variable type record has a record header. The header contains the record length and the length of the preceding record. The system writes the header when the record is written to the file. The system uses the header information for positioning the file. The record header is not transferred to the working storage area with the rest of the record when a record is read.

The end-of-partition (EOP) delimiter for the variable record type is a record header that has record length of zero and end-of-partition flag set.

### ANSI-Fixed Length Record Type

Fixed length records all have the same length. The fixed length is the number of bytes specified by the value of the MAXIMUM_RECORD_LENGTH attribute. If the number of bytes to be written as a record is less than the maximum record length, the system pads the record.

The fixed length record type supports partial record I/O and provides efficient storage for records of a constant length.

Also, when used with user-specified blocking, the fixed length record type supports data interchange between different systems because it is an American National Standard Institute standard record type.

The fixed length record type does not support file partitioning.

### Undefined Record Type

A file with undefined record type is viewed by the system as an unstructured byte string. The system transfers the number of bytes specified to or from the working storage area. The system does not impose any structure on the file other than file blocking.

The undefined record type supports tape files with unblocked variable length records, data interchange between different systems, and partial record I/O.

The undefined record type does not support file partitioning.

## File Attributes

A file attribute is a characteristic of the file. File attributes describe the structure of the file and define processing limitations for the file.

File attributes can be separated into three categories: preserved, temporary, and returned.

Preserved attributes determine file structure. The values of these attributes are defined when the file is created and kept for the life of the file. The values of some preserved attributes can be changed with the CHANGE_FILE_ATTRIBUTES (CHAFA) command.

Temporary attributes determine how the job processes the file. You set temporary file attributes with an ATTACH_FILE command. The values of temporary attributes are discarded when the file is detached. An implicitly attached file is detached at the end of a command.

Returned attributes are set by the file interface. The values of returned attributes can be fetched for information on the current state of the file.

The following lists the most common file attributes. One asterisk (*) indicates attributes that apply only to indexed sequential files. Two asterisks (**) indicate attributes that apply only to direct access files. Three asterisks (***) indicate attributes that apply to both indexed sequential and direct access (but not to any other file organization).

See the SCL System Interface Usage manual for a complete list of file attributes.

ACCESS_MODE
CHARACTER_CONVERSION
COLLATE_TABLE_NAME***
DATA_PADDING*
EMBEDDED_KEY***
FILE_CONTENTS
FILE_LIMIT
FILE_ORGANIZATION
FILE_PROCESSOR
FILE_STRUCTURE
HASHING_PROCEDURE_NAME**
INDEX_PADDING*
INITIAL_HOME_BLOCK_COUNT**
INTERNAL_CODE
KEY_LENGTH***
KEY_POSITION***
KEY_TYPE***
LOCK_EXPIRATION_TIME
MAXIMUM_BLOCK_LENGTH
MAXIMUM_RECORD_LENGTH
MESSAGE_CONTROL***
OPEN_POSITION
PADDING_CHARACTER
PAGE_FORMAT
PAGE_LENGTH
PAGE_WIDTH
RECORD_TYPE

## ACCESS_MODE (AM)

The ACCESS_MODE attribute specifies the set of access permissions for the file. (Temporary attribute) The usual way to specify ACCESS_MODE is with an ATTACH_FILE command. You can also specify ACCESS_MODE in the CREATE_FILE_PERMIT command and the CREATE_CATALOG_PERMIT command (both described in chapter 4, Common NOS/VE commands). ACCESS_MODE can be a subset of the following values:

| | |
|---|---|
| READ | File can be read. |
| WRITE | Data can be written to file (implies APPEND, MODIFY, SHORTEN). |
| APPEND | Data can be added at the end of the file. |
| MODIFY | Data in file can be altered. |
| SHORTEN | Data can be deleted from file. |
| EXECUTE | File can be executed. |
| NONE | File cannot be accessed until file access is restored by an ATTACH_FILE command. |

Omission for a new temporary file causes the READ and WRITE permission to be used. Omission for an old file causes READ and/or WRITE to be used, depending (usually) on the kind of processing required for the job.

Omission for a permanent file scheduled for job access using the ATTACH_FILE command causes the ACCESS_MODE specified on that command to be used as qualified (usually) by the kind of processing required for the job.

## CHARACTER_CONVERSION (CC)

The CHARACTER_CONVERSION attribute specifies whether conversion between the internal character code of a file and ASCII should be performed. (Preserved attribute)

The CHARACTER_CONVERSION attribute can have the following values:

| | |
|---|---|
| TRUE | Conversion is performed |
| FALSE | Conversion is not performed |

Conversion of a tape file is performed by the file interface on a block by block basis as the file is accessed. File access routines do not perform conversion of disk files and do not use the CHARACTER_CONVERSION attribute. The attribute is available to utilities that do conversion on disk files.

## COLLATE_TABLE_NAME (CTN)

The COLLATE_TABLE_NAME attribute specifies the name of the collation table used for indexed sequential and direct access files. A collation is required only if the KEY_TYPE attribute specifies COLLATED. The collation table establishes respective weights for characters, and thereby effects the ordering of records in the file. The table exists as a program in a library or as an entry point already in memory. Once a file is open, COLLATE_TABLE_NAME cannot be set. (Preserved attribute)

You can specify one of the following predefined collation tables:

```
OSV$DISPLAY63_FOLDED
OSV$DISPLAY64_FOLDED
OSV$COBOL6_FOLDED
OSV$ASCII6_FOLDED
OSV$EBCDIC6_FOLDED
OSV$EBCDIC
OSV$DISPLAY63_STRICT
OSV$DISPLAY64_STRICT
OSV$COBOL6_STRICT
OSV$ASCII6_STRICT
OSV$EBCDIC6_STRICT
```

The collation tables are listed in appendix C. For information about using the tables, see the section Predefined Collation Tables in chapter 11.

If the predefined tables are not appropriate for your use, you can create your own collate tables. For information about creating collate tables, see the FORTRAN Language Definition Usage manual.

## DATA_PADDING (DP)

The DATA_PADDING attribute specifies the percentage of space the file interface is to leave when writing records to data blocks at file creation time. This attribute applies only to indexed sequential files. (Preserved attribute)

DATA_PADDING can be set to an integer from 0 to 99.

Omission for a new indexed sequential file causes 0 (no padding) to be used.

## EMBEDDED_KEY (EK)

The EMBEDDED_KEY attribute is used by the file interface to locate the primary key. This attribute applies to indexed sequential and direct access files. (Preserved attribute)

EMBEDDED_KEY can be set as follows:

YES  Primary key is located in the record.

NO  Primary key is stored separately from the record.

Omission for a new indexed sequential or direct access file causes YES to be used.

## FILE_CONTENTS (FC)

The FILE_CONTENTS attribute specifies the type of data contained in the file. (Preserved attribute)

The valid options for FILE_CONTENTS include:

| | |
|---|---|
| UNKNOWN | File contents are unknown. |
| OBJECT | File is an object module. |
| LIST | File contains character data for printing; the first character of each record is interpreted as a print format control character. |
| LEGIBLE | File contains character data. |

Omission for a new file causes UNKNOWN to be used. Omission for an old file causes the preserved value to be used.

## FILE_LIMIT (FL)

The FILE_LIMIT attribute specifies the maximum length of the file in bytes. (Preserved attribute)

FILE_LIMIT can be set to an integer between 1 and the system maximum of 150,000,000.

Omission for a new file causes the system maximum of 150,000,000 to be used. Omission for an old file causes the preserved value to be used.

An error occurs if the length of the file exceeds the FILE_LIMIT.

## FILE_ORGANIZATION (FO)

The FILE_ORGANIZATION attribute specifies the organization of the file. (Preserved attribute)

FILE_ORGANIZATION can be set to the following values:

| | |
|---|---|
| SEQUENTIAL (SQ) | Records are stored and retrieved in the order they are presented to the system. |
| BYTE_ADDRESSABLE (BA) | Records are stored and retrieved according to the byte address at the start of record. |
| INDEXED_SEQUENTIAL (IS) | Records are stored according to primary key value. Records are retrieved according to primary key value or record position. |
| DIRECT_ACCESS (DA) | Records are stored randomly according to the hashed value of a primary key. Records are retrieved according to primary key value or record position. |

Omission for a new file causes SEQUENTIAL to be used. Omission for an old file causes the preserved value to be used.

## FILE_PROCESSOR (FP)

The FILE_PROCESSOR attribute specifies the name of the processor for the file. This attribute qualifies the FILE_CONTENTS attribute and is used by NOS/VE to verify correct usage of the file. (Preserved attribute)

FILE_PROCESSOR can be set to the following values:

| | |
|---|---|
| ADA | ADA compiler |
| APL | APL compiler |
| ASSEMBLER | NOS/VE assembler |
| BASIC | BASIC compiler |
| C | C compiler |
| COBOL | COBOL compiler |
| CYBIL | CYBIL compiler |
| DEBUGGER | Debug utility |
| FORTRAN | FORTRAN compiler |
| PASCAL | Pascal compiler |
| PL1 | PL1 compiler |
| PPU_ASSEMBLER | NOS PP assembler |
| PROLOG | PROLOG compiler |
| SCL | System Command Language interpreter |
| SCU | Source Code Utility text |
| UNKNOWN | Processor is unknown |
| VX | File processor associated with VX/VE |

Omission for a new file causes UNKNOWN to be used. Omission for an old file causes the preserved value to be used.

## FILE_STRUCTURE (FS)

The FILE_STRUCTURE attribute specifies the structure of the file. This attribute qualifies the FILE_CONTENTS and FILE_PROCESSOR attributes and is used by NOS/VE to verify correct usage of the file.

The valid options for FILE_STRUCTURE include:

| | |
|---|---|
| UNKNOWN | Structure of the file is unknown |
| DATA | Data file |
| LIBRARY | Library file |

Omission for a new file causes UNKNOWN to be used. Omission for an old file causes the preserved value to be used.

## HASHING_PROCEDURE_NAME (HPN)

The HASHING_PROCEDURE_NAME attribute specifies the name of a hashing procedure to be used for a direct access file. The default procedure is the one provided by the operating system (AMP$SYSTEM_HASHING_PROCEDURE).

## INDEX_PADDING (IP)

The INDEX_PADDING attribute specifies the percentage of empty space the file interface must leave in each index block when an indexed sequential file is created. This attribute applies only to indexed sequential files. (Preserved attribute)

INDEX_PADDING can be set to a value from 0 through 99.

Omission for a new indexed sequential file causes 0 (no padding) to be used.

## INITIAL_HOME_BLOCK_COUNT (IHBC)

The INITIAL_HOME_BLOCK_COUNT attribute specifies the number of home blocks created when the file is created. It must be a value in the range 1 through (2**42)-1. This attribute must be specified when creating a direct access file.

## INTERNAL_CODE (IC)

The INTERNAL_CODE attribute specifies the internal code used to represent data in the file. (Preserved attribute)

INTERNAL_CODE can be set to the following values:

| | |
|---|---|
| A6 | NOS 6/12-bit display code, which results from using ASCII mode on NOS (ASCII 128-character set) |
| A8 | NOS 12-bit ASCII code (ASCII 128-character set) |
| ASCII | NOS/VE 7-bit ASCII code (ASCII 128-character set) |
| D63 | NOS 6-bit ASCII display (CDC 63-character set) |
| D64 | NOS 6-bit display code (CDC 64-character set) |

Omission for a new file causes ASCII to be used. Omission for an old file causes the preserved value to be used.

## KEY_LENGTH (KL)

The KEY_LENGTH attribute specifies the length of the primary key being defined for the new indexed sequential or direct access file. You must set KEY_LENGTH before a new file can be opened. (Preserved attribute)

KEY_LENGTH can be set from 1 to 255. For files with embedded keys, the value of KEY_LENGTH cannot be greater than the value of MINIMUM_RECORD_LENGTH.

There is no default; you must set KEY_LENGTH when creating an indexed sequential file.

## KEY_POSITION (KP)

The KEY_POSITION attribute specifies the byte offset into each record where the primary key begins. KEY_POSITION is ignored for indexed sequential or direct access files with nonembedded keys. This attribute applies only to indexed sequential and direct access files. (Preserved attribute)

KEY_POSITION can be set to a value from zero to MAXIMUM_RECORD_LENGTH; however, KEY_POSITION + KEY_LENGTH must be less than or equal to the value of MINIMUM_RECORD_LENGTH. (The value of KEY_POSITION + 1 defines the first character of the key. For example, if KEY_POSITION equals three, the key begins with the fourth character of the record.) The absolute maximum value for KEY_POSITION is 65,535.

Omission for a new indexed sequential and direct access file causes zero to be used.

## KEY_TYPE (KT)

The KEY_TYPE attribute specifies the type of the primary key being defined for a new indexed sequential file. The KEY_TYPE attribute value for direct-access files is always UNCOLLATED. (Preserved attribute)

KEY_TYPE can be set to one of the following three values:

| | |
|---|---|
| UNCOLLATED (UC) | The key is a string of alphanumeric characters. Uncollated keys are compared using the ASCII collating sequence. |
| INTEGER (I) | The key is an integer from 1 to 8 bytes long. Integer keys are compared as signed integers. |
| COLLATED (C) | The key is a string of alphanumeric characters. Collated keys are compared using a user-supplied collation table. If you specify this type of key, you must supply a collation table. Several predefined tables are available. See information about predefined collation tables in chapter 11, General Facilities for Migrating Files. |

Omission for a new indexed sequential or direct access file causes UNCOLLATED to be used. Omission for an old indexed sequential or direct access file causes the preserved value to be used.

## LOCK_EXPIRATION_TIME (LET)

The LOCK_EXPIRATION_TIME attribute specifies the number of milliseconds between the time a lock is granted and the time that it could expire. It must be an integer in the range 0 through 604,800,000 (one week). The default is 60,000 milliseconds. (Preserved attribute)

## MAXIMUM_BLOCK_LENGTH (MAXBL)

The MAXIMUM_BLOCK_LENGTH attribute specifies the maximum length (from 0 to 65,497) in bytes of a block in a file. This attribute is ignored when system-specified blocking is requested. (Preserved attribute)

For keyed files, block length is set at creation time and does not change thereafter. If MAXIMUM_BLOCK_LENGTH is specified at creation time, the actual block length is the nearest higher number in the list 2,048, 4,096, 8,192, 16,384, 32,768, 65,536.

MAXIMUM_BLOCK_LENGTH for a tape file is 4,128 bytes.

Records are packed into blocks according to ANSI 1978 standards. For disk files, transfers between central memory and the device are in multiples of one or more blocks. For a tape file, MAXIMUM_BLOCK_LENGTH determines the size of the physical record written to a tape volume.

Omission for a new file causes 4,128 bytes to be used. Omission for an old file causes the preserved value to be used.

## MAXIMUM_RECORD_LENGTH (MAXRL)

The MAXIMUM_RECORD_LENGTH attribute specifies the maximum length in bytes of a record in the file. (Preserved attribute)

The MAXIMUM_RECORD_LENGTH attribute is used only for sequential and byte addressable files containing fixed length records (ANSI-fixed), and for all indexed sequential files.

MAXIMUM_RECORD_LENGTH can be set to a value between 1 and 2**42 -1 bytes.

Omission for a new sequential or byte addressable file causes 256 to be used. Omission for an old sequential or byte addressable file causes the preserved value to be used. Omission for a new indexed sequential or direct access file causes an error to occur because MAXIMUM_RECORD_LENGTH must be set before an indexed sequential or direct access file can be created. Omission for an old indexed sequential or direct access file causes the preserved value to be used.

## MESSAGE_CONTROL (MC)

The MESSAGE_CONTROL attribute controls the listing of trivial error messages, statistics, and informative messages on the $ERRORS file (fatal error messages are always logged). This attribute applies only to indexed sequential or direct access files. (Temporary attribute) You set the MESSAGE_CONTROL attribute with the ATTACH_FILE command.

MESSAGE_CONTROL can be a set of the following values:

| | |
|---|---|
| TRIVIAL_ERRORS (T) | Trivial error messages are logged. |
| MESSAGES (M) | Informative messages are logged. |
| STATISTICS (S) | Statistical messages are logged. |
| NONE (N) | Only fatal error messages are logged. |

Omission for a new indexed sequential or direct access file causes NONE to be used.

## OPEN_POSITION (OP)

The OPEN_POSITION attribute specifies the positioning to occur when the file is opened. (Temporary attribute) You set the OPEN_POSITION attribute with the ATTACH_FILE command.

| | |
|---|---|
| $BOI | File is opened at beginning-of-information. |
| $ASIS | File is opened with no positioning. |
| $EOI | File is opened at end-of-information. |

If the OPEN_POSITION for a file is specified using both the file reference and the ATTACH_FILE command, the value specified in the ATTACH_FILE command is used.

Omission for a file causes the value specified by a file reference to be used. Omission from both the ATTACH_FILE command and the file reference causes $EOI to be used for the OUTPUT file and $BOI to be used for all other files.

## PADDING_CHARACTER (PC)

The PADDING_CHARACTER attribute specifies the padding character used to pad short ANSI-fixed records to the MAXIMUM_RECORD_LENGTH defined for the file. (Preserved attribute)

Omission for a new file causes the space character to be used. Omission for an old file causes the preserved value to be used.

## PAGE_FORMAT (PF)

The PAGE_FORMAT attribute specifies the frequency and separation of titling in a legible file. The attribute is used only by the file access routines if the file is associated with a terminal. It is used by other services to prepare files for printing. (Preserved attribute)

PAGE_FORMAT can be set to the following values:

| | |
|---|---|
| CONTINUOUS | Title appears once at the beginning of the file. |
| BURSTABLE | Title and page number appear at the top of each page of the file. |
| NON_BURSTABLE | Title and page number separated from other data by a triple space rather than by forcing top of form as in the burstable selection. |

Omission for a new terminal file causes CONTINUOUS to be used. Omission for a new nonterminal file causes BURSTABLE to be used. Omission for an old file causes the preserved value to be used.

## PAGE_LENGTH (PL)

The PAGE_LENGTH attribute specifies the number of lines to be written on a printed page. The attribute is used only by the file access routines if the file is associated with a terminal. Also, the attribute is used by other services to prepare files for printing. (Preserved attribute)

Omission for a new file causes 60 lines per printed page to be used. Omission for an old file causes the preserved value to be used.

## PAGE_WIDTH (PW)

The PAGE_WIDTH attribute specifies the number of characters to be written on a printed line. The attribute is used only by the file access routines if the file is associated with a terminal. Also, the attribute is used by other services to prepare files for printing. (Preserved attribute)

Omission for a new file causes the value of the PAGE_WIDTH terminal attribute to be used for a terminal file and 132 characters per printed line to be used for a nonterminal file.

## RECORD_TYPE (RT)

The RECORD_TYPE attribute specifies the type of record in the file. (Preserved attribute)

RECORD_TYPE can be set to the following values:

| | |
|---|---|
| VARIABLE (V) | CDC-variable |
| FIXED (F) | ANSI-fixed |
| UNDEFINED (U) | Undefined |

Normally a file is of the kind that is accessed at the record level. Such a file is called a record access file. If you omit the RECORD_TYPE attribute for a new record access file, the VARIABLE value is used. Omission of the RECORD_TYPE attribute for the other kinds of new disk files causes UNDEFINED to be used. Omission for an old file causes the preserved value to be used.

For more information about these record types, see the discussion about NOS/VE record types near the beginning of this chapter.

## File Attribute Defaults Used by FORTRAN Programs

The following paragraphs describe the structure of the files read and written by FORTRAN. All files read and written by FORTRAN input/output statements, as well as the files read and written by the FORTRAN compiler, are processed through internal file interface routines.

### Execution-Time Input/Output

All input and output between a file referenced in a program and the external storage device is under control of the internal file interface routines. These routines encompass sequential, indexed sequential, direct access, and byte addressable file organizations.

Each NOS/VE file is described by an internally maintained table of file attributes. File processing is governed by values the FORTRAN compiler places in this table. Some of the values of the file attributes are permanent for the life of the file; others can be changed by a SET_FILE_ATTRIBUTES command, a CHANGE_FILE_ATTRIBUTES command, or by parameters in the PROGRAM and OPEN statements.

More details about the files FORTRAN uses is given in the following topics:

File and record definitions

File structure

Defaults for file attributes

Overriding FORTRAN defaults

Sequential I/O

FORTRAN direct access I/O

#### File and Record Definitions

A file is a collection of records. It is the largest collection of information that can be referenced by a file name. A file begins at beginning-of-information and ends at end-of-information. A record is a contiguous group of bytes within a file and is read or written as a single unit. A record is read or written by:

One execution of an unformatted READ or WRITE statement.

A formatted, list directed, or namelist READ or WRITE statement. (A single execution of these statements can transmit more than one record.)

One call to READMS or WRITMS.

One execution of a BUFFER IN or BUFFER OUT.

The record types are:

V CDC-variable length records

F ANSI-fixed length records

U Undefined length records

FORTRAN uses the V and F record types.

### File Structure

FORTRAN sets certain file attributes, depending on the nature of the input/output operation and its associated file structure. Most file attributes are preserved for the life of a file. After a file is created (that is, after the file is opened for the first time), the preserved file attributes that define the structure of the file cannot be changed. The file attributes for the various types of FORTRAN input/output are shown in tables that follow. The attributes that can be overridden by a SET_FILE_ATTRIBUTES (SETFA) command or CHANGE_FILE_ATTRIBUTES (CHAFA) command prior to file creation are indicated by two crosses. The attributes that can be overridden prior to any open of the file are indicated by one cross. Files connected to $INPUT or $OUTPUT retain the attributes of $INPUT or $OUTPUT regardless of SETFA or CHAFA specifications.

Tables 10-4 and 10-5 provide more information about the defaults for file attributes for files involved in formatted sequential I/O, mass storage I/O, unformatted sequential I/O, FORTRAN direct access I/O, and buffer I/O.

Table 10-4. Defaults for File Attributes for Formatted and Unformatted Sequential I/O

| File Attribute | Formatted Sequential I/O | Unformatted Sequential I/O |
|---|---|---|
| MAXIMUM_RECORD_LENGTH | RECL= in OPEN statement | RECL= in OPEN statement |
| OPEN_POSITION | $ASIS† | $ASIS† |
| ACCESS_MODE | R/W/A/M† | R/W/A/M† |
| FILE_ORGANIZATION | SQ | SQ |
| RECORD_TYPE | V†† | V†† |
| PADDING_CHARACTER | not applicable | not applicable |
| PAGE_WIDTH | 132 characters (nonconnected file)†† | not applicable |
| | Value of the PAGE_WIDTH terminal attribute (connected file) | not applicable |

†Can be overridden by SET_FILE_ATTRIBUTES command prior to any open
††Can be overridden by SET_FILE_ATTRIBUTES command prior to file creation

| | |
|---|---|
| $ASIS | Current file position |
| R/W/A/M | READ/WRITE/APPEND/MODIFY |
| SQ, BA | Sequential, Byte addressable |
| V, F, U | Variable-length, Fixed-length, Undefined |

Table 10-5. Defaults for File Attributes for Buffer, Mass Storage, and FORTRAN Direct Access I/O

| File Attribute | Buffer I/O | Mass Storage I/O | Direct Access I/O |
|---|---|---|---|
| MAXIMUM_RECORD_LENGTH | not applicable | not applicable | RECL= in OPEN statement |
| OPEN_POSITION | $ASIS† | not applicable | not applicable |
| ACCESS_MODE | R/W/A/M† | R/W/A/M† | R/W/A/M† |
| FILE_ORGANIZATION | SQ | BA | BA |
| RECORD_TYPE | V†† | U | F |
| PADDING_CHARACTER | not applicable | not applicable | blank†† |
| PAGE_WIDTH | not applicable | not applicable | not applicable |

†Can be overridden by SET_FILE_ATTRIBUTES command prior to any open
††Can be overridden by SET_FILE_ATTRIBUTES command prior to file creation

$ASIS Current file position
R/W/A/M READ/WRITE/APPEND/MODIFY
SQ, BA Sequential, byte addressable
V, F, U Variable-length, fixed-length, undefined

### Overriding FORTRAN Default Values

The SET_FILE_ATTRIBUTES command provides a means of overriding file attributes compiled into a program, and consequently, a means to change processing normally supplied for FORTRAN input/output. In particular, this command enables you to read or create a file with attributes that are different from those supplied by default.

The file attributes specified on a SET_FILE_ATTRIBUTES command are established when a file is created (that is, the first time it is opened).

### SET_FILE_ATTRIBUTES Example for FORTRAN

The following program opens and writes a file named AFILE:

```
PROGRAM ABC
OPEN (FILE='AFILE', UNIT=1)
WRITE (1,100) A, B, C
.
.
.
```

The following SET_FILE_ATTRIBUTES command, specified before the program is executed, overrides the default maximum record length of 150 characters:

```
/set_file_attributes file=afile maximum_record_length=100
```

A MAXIMUM_RECORD_LENGTH specification in a SET_FILE_ATTRIBUTES command prior to program execution takes precedence over a record length specification in an OPEN or PROGRAM statement. For FORTRAN direct access files, if MAXIMUM_RECORD_LENGTH is specified in a SET_FILE_ATTRIBUTES command prior to execution, and if an OPEN statement specifies a different record length, a fatal error is issued.

For more information about the SET_FILE_ATTRIBUTES command see the discussion of the command in chapter 4, Common Commands. For more information about the individual file attributes (specified in the SET_FILE_ATTRIBUTES command), see the discussion of file attributes earlier in this chapter.

### Sequential Input/Output

The sequential READ and WRITE statements, namelist I/O statements, list directed I/O statements, and buffer I/O statements process sequential files with V type records. The record type can be overridden by a SET_FILE_ATTRIBUTES command before execution.

The BACKSPACE, REWIND, and ENDFILE operations are valid only for sequential files with V type records. BACKSPACE skips backward (toward beginning-of-information) one record. (The file is positioned before the record just read or written.) REWIND positions a file at beginning-of-information. ENDFILE writes an end-of-partition boundary.

When an end-of-partition is encountered during a read, the ERR= specifier and EOF function return end-of-file status. If the end-of-partition does not coincide with end-of-information, you can continue reading the same file until the end-of-information is encountered.

For F and U type records, the EOF and UNIT functions return end-of-file status only at end-of-information.

### FORTRAN Direct Access Input/Output

Direct access input/output statements process byte addressable files with F type records. F is the only record type permitted for direct access input/output.

The file positioning statements (BACKSPACE, REWIND, and ENDFILE) cannot be used with FORTRAN direct access files.

### Compile-Time Input/Output

The FORTRAN compiler reads a source input file and produces up to three output files: a binary object file, an output listing file, and an error listing file. The compiler expects the input source file to have a certain structure, and it produces output files that have specific structures. Table 10-6 describes the attributes of the compiler input and output files.

Table 10-6. Compile-Time File Structure

| File Attribute | Source Input File | Compiler Output Listing File | Error File | Binary Object File |
|---|---|---|---|---|
| FILE_ORGANIZATION | SQ | SQ | SQ | SQ |
| FILE_STRUCTURE | DATA† | DATA† | DATA† | DATA |
| FILE_CONTENTS | LEGIBLE† | LEGIBLE† (Interactive)<br>LIST (Batch) | LIST† | OBJECT |
| RECORD_TYPE | V† | V† | V† | V |

†Can be overridden by SET_FILE_ATTRIBUTES command prior to file creation

SQ Sequential
V Variable-length

## File Attribute Defaults Used by COBOL Programs

All COBOL file input/output requests are implemented by file interface routines. Before opening a file, the compiler generates information about the file and passes the information to the operating system.

All the file attributes needed to process the file are set automatically based on source code. This section outlines how the source code and setting of file attributes are related.

### File Attributes and Associated COBOL Source Code

Source code statements set certain file attributes. Table 10-7 lists the COBOL statements and associated attributes. For detailed information about setting file attributes from COBOL source code, see the COBOL for NOS/VE Usage manual.

Table 10-7. File Attribute and Associated COBOL Source Code

| File Attribute | COBOL Source Code That Sets the Attribute |
|---|---|
| ACCESS_MODE | OPEN phrase (can depend on ATTACH_FILE command) |
| BLOCK_TYPE | File on tape, LABEL RECORDS STANDARD, and BLOCK CONTAINS |
| CHARACTER_CONVERSION | ALPHABET and CODE-SET |
| EMBEDDED_KEY | RECORD KEY |
| FILE_CONTENTS | WRITE ADVANCING REPORTS |
| FILE_ORGANIZATION | ORGANIZATION clause |
| FILE_PROCESSOR | Always set to COBOL |
| FILE_STRUCTURE | See FILE_CONTENTS |
| INTERNAL_CODE | ALPHABET and CODE-SET |
| KEY_LENGTH | PICTURE clause for key and USAGE |
| KEY_POSITION | Location of key |
| KEY_TYPE | Class and USAGE |
| MAXIMUM_BLOCK_LENGTH | BLOCK CONTAINS and LABEL RECORDS and Record Descriptions |
| MAXIMUM_RECORD_LENGTH | RECORD CONTAINS or Record Descriptions |
| MINIMUM_RECORD_LENGTH | RECORD CONTAINS or Record Descriptions |
| OPEN_POSITION | OPEN phrase |
| PAGE_LENGTH | LINAGE or Report Writer clauses |
| PAGE_WIDTH | Record Description Entries (Report Writer, Linage, or PRINTF=YES files only) |
| RECORD_TYPE | RECORD clause or Record Descriptions |

## Special Cases

In most cases, setting file attributes from source code is straightforward. However, you should be careful of the defaults that result from the following four clauses:

BLOCK CONTAINS clause

CODE-SET clause

LINAGE clause

RECORD clause

### BLOCK CONTAINS Clause

If you specify the BLOCK CONTAINS clause for a sequential file that is a standard system file (the implementor-name begins with a $), the clause is ignored and BLOCK_TYPE (BT) is set to SYSTEM_SPECIFIED.

If you specify the BLOCK CONTAINS clause for a sequential file that is not a standard system file, BLOCK_TYPE (BT) is set to USER_SPECIFIED.

### CODE-SET Clause

If you specify CODE-SET for a sequential file, INTERNAL_CODE (IC) is set to the alphabet-name specified in the ALPHABET and CODE-SET clause, and CHARACTER_CONVERSION (CC) is set to TRUE.

### LINAGE Clause

If you specify the LINAGE clause, FILE_CONTENTS, FILE_STRUCTURE, PAGE_FORMAT, and PAGE_LENGTH are set as follows:

FILE_CONTENTS (FC) is set to LIST.

FILE_STRUCTURE (FS) is set to DATA.

PAGE_FORMAT (PF) is set to CONTINUOUS if LINES AT TOP or LINES AT BOTTOM is specified; otherwise PAGE_FORMAT (PF) is set to BURSTABLE.

PAGE_LENGTH is set to integer-1 or the value of data-name-1.

### RECORD Clause

If you specify the RECORD clause for a byte addressable file, RECORD_TYPE is set to U type records.

If you specify the RECORD clause for a relative file, RECORD_TYPE is set to U type records; however, the records are processed as fixed length records.

If you specify the RECORD clause for a sequential file that is a standard system file (implementor-name begins with $), RECORD_TYPE is set to V type records.

If you specify the RECORD clause for a sequential file that is not a standard system file or for an indexed sequential or direct access file, RECORD_TYPE is set to V or F type records as follows:

- RECORD_TYPE is set to V type records, if any of the following conditions are true:

  The RECORD CONTAINS ... DEPENDING ON phrase is specified in the FD.

  PRINT_FILE = TRUE is specified in the USE clause.

  TRUNCATE_SPACES is specified in the USE clause.

- RECORD_TYPE is set to F type records if none of the above conditions are true.

# General Facilities for Migrating Files 11

# General Facilities for Migrating Files 11

Migrating files from NOS to NOS/VE requires converting files from a NOS format to a NOS/VE format and transferring from NOS to NOS/VE, or vice versa. Both program files and data files require migration.

First this discussion provides an overview of the NOS/VE file migration facilities. Then the discussion presents the following facilities:

GET_FILE (GETF) and REPLACE_FILE (REPF) commands

Transfers data files from NOS to NOS/VE, or vice versa.

SCU conversion commands

Migrates Update and Modify source library files.

The permanent file transfer facility

Transfers character data files from NOS to NOS/VE and vice versa.

File Management Utility

Migrates and converts files.

Predefined Collation Tables

Specifies user-selected collating sequence for indexed sequential files.

## Overview

Migrating text files is handled easily with the GET_FILE (GETF) and REPLACE_FILE (REPF) commands and with the permanent file transfer facility (PTF). Use GET_FILE, REPLACE_FILE, and the permanent file transfer facility with text files only. Using these migration methods on other kinds of files can produce undesired results.

GET_FILE converts and transfers files from NOS to NOS/VE; REPLACE_FILE does the reverse operation. The text files can be either data or program source code. However, on NOS, the files must be sequential with block type C and record type Z (for zero-byte). Files created with a text editor, COBOL DISPLAY statement, or FORTRAN PRINT or formatted WRITE statements are of this type. The following diagram shows using the GET_FILE command.


The permanent file transfer facility (PTF) allows you to initiate a job on a local system to access a file on a remote system. In particular, PTF enables NOS/VE systems to transfer character data files to NOS systems and vice versa.

To transfer files the following criteria must be met:

- The local system and the remote system must be connected via CDCNET.
- Both systems must support PTF.
- You must provide appropriate user validation information and commands on both systems.

Transferring program source code is the only way to migrate programs from NOS to NOS/VE. No object code generated to run on NOS will run on NOS/VE. To migrate programs, follow the migration methods for converting the program source code (see chapter 13, Approaching COBOL and FORTRAN Program Migration). Either develop the source code on NOS and transfer your file to NOS/VE using the GET_FILE command or the permanent file transfer facility, or develop your source code on NOS/VE by using the SCL file editor. Transfer the Update or Modify source library files by using the NOS/VE Source Code Utility (SCU) conversion commands.

Binary data files or character data files in formats other than the sequential CZ format can be transferred to NOS/VE in several ways.

First, you could write a program to convert the file to character data in CZ format. Specifying the following FILE command and including it in the command stream prior to program execution designates a CZ file:

```
FILE,lfn,BT=C,RT=Z,FL=length  <-- lfn is local file name.
```

Then transfer the sequential CZ file to NOS/VE with the GET_FILE command and recreate your file in NOS/VE format with a program that reads the character data and writes another file in the appropriate format. However, this way requires lots of coding and a number of steps.

Secondly, you can use the File Management Utility (FMU) to convert and transfer your files from NOS to NOS/VE. You use FMU from the NOS/VE state, issue interstate commands to attach and describe your NOS file, and issue NOS/VE commands to convert and transfer your file to NOS/VE.

Alternately, for files read by FORTRAN, you could migrate the files by using the FORTRAN File Migration Aid (FMA) in the input/output type is supported. FORTRAN FMA migrates a file as it is read by a NOS/VE FORTRAN program. FMA supports several kinds of FORTRAN input/output. See chapter 12 for the description of FORTRAN FMA.

## GET_FILE (GETF) and REPLACE_FILE (REPF)

The NOS/VE command GET_FILE transfers a copy of a NOS permanent file (direct or indirect access) to NOS/VE and converts it to NOS/VE format. The file can be a text or binary file; however, the GET_FILE discussion shows transerring only text files. On NOS, the text files must have Z-type (zero-byte) records. (Files created by text editors have Z-type records.)

The NOS/VE command REPLACE_FILE transfers a file from NOS/VE to a NOS direct or indirect access permanent file. If the file exists, REPLACE_FILE rewrites the file. If the file does not exist, REPLACE_FILE performs a DEFINE operation. REPLACE_FILE can handle either a text or binary file; however, the REPLACE_FILE discussion shows transferring only text files.

See chapter 4, Common NOS/VE Commands, for more information about using these commands.

## SCU Conversion Commands

NOS/VE provides commands to directly convert Update and Modify source libraries to Source Code Utility (SCU) library format. The commands are CONVERT_UPDATE_TO_SCU (CONUTS) and CONVERT_MODIFY_TO_SCU (CONMTS).

To convert an Update or Modify library to SCU format, you would first transfer the file from NOS to NOS/VE using the GET_FILE command. The GET_FILE command must specify the DATA_CONVERSION (DC) parameter value DC=B60 (which specifies a binary transfer that places each 60-bit NOS word in the right most bits of each 64-bit NOS/VE word). You then enter the appropriate conversion command to convert the library. For example, the following commands convert the Update library file OLDPL to an SCU library on file BASE. (OLDPL is the default Update library name.)

```
/get_file  to=oldpl  data_conversion=b60
/convert_update_to_scu  result=base
```

For details on the limitations of the conversion process, see the SCL Source Code Management Usage manual.

## The Permanent File Transfer Facility (PTF)

The permanent file transfer facility consists of two partner applications: the permanent file client application and the permanent file server application. They are on the local system and the remote system, respectively. The remote system application is called the server application because you direct it through the client application.

When NOS is the local system, PTF uses the MFLINK control statement to transfer permanent files. MFLINK must send instructions, called permanent file server directives, to the remote system to tell it what to do for the file transfer. The directives are statements/commands in the language of the remote operating system. For example, permanent file server directives for a remote NOS/VE system are SCL commands.

The following is a commonly used format of MFLINK:

```
MFLINK,lfn,ST=lid
```

The lfn parameter is the name of the NOS temporary file to be used in the file transfer.

The lid parameter is the logical identifier of the remote system. Obtain this identifier from your site administrator.

This format tells NOS that the permanent file server directives for the remote system are on the lines immediately following the MFLINK control statement.

The Remote Host Facility Usage manual gives the complete MFLINK format and shows how to use MFLINK to perform a variety of remote file accesses.

### Transferring Files From NOS to NOS/VE By Using MFLINK on NOS

To transfer a NOS file on one mainframe to a NOS/VE file on another mainframe, you first log into the NOS system, the local system. Next, enter the MFLINK control statement on NOS. Finally, enter the permanent file server directives for NOS/VE, the remote system. These directives must include a NOS/VE login and the SCL RECEIVE_FILE command. The RECEIVE_FILE command causes a file in the remote NOS/VE system to accept a file sent from another system. In particular, you use the RECEIVE_FILE command to send a file from your local NOS system on one mainframe to the remote NOS/VE system on another mainframe.

The SCL System Interface Usage manual fully describes RECEIVE_FILE. A commonly used format for RECEIVE_FILE follows:

```
RECEIVE_FILE  or
RECF
    FILE=file
```

The FILE or F parameter specifies the name of the file on the remote system that is to receive the file from your local system. This file must be a permanent file residing on the remote system.

The following example transfers temporary file AFILE in the local NOS system to permanent file $USER.TEXT_COPY in a remote NOS/VE system. In the example, $USER.TEXT_COPY has to be created.

```
MFLINK,AFILE,ST=FM2
```

AFILE names the NOS temporary file to be transferred. The remote NOS/VE system has logical identifier FM2.

```
*LOGIN USER=USER2  PW=PASSWD2  FN=FM2
```

Log in for the remote NOS/VE system with user name USER2, password PASSWD2, and family name FM2.

The * is the NOS prompt for the permanent file server directives.

```
*CREATE_FILE  FILE=$USER.TEXT_COPY
```

Explicitly creates the permanent NOS/VE file $USER.TEXT_COPY to receive AFILE.

```
*RECEIVE_FILE  FILE=$USER.TEXT_COPY
```

Causes $USER.TEXT_COPY in the remote NOS/VE system to receive AFILE from the local NOS system.

```
* (Carriage return)
```

Indicates end of the permanent file server directives.

## Transferring Files From NOS/VE to NOS By Using MFLINK on NOS

To transfer a NOS/VE file on one mainframe to a NOS file on another mainframe, you first log into the NOS system, the local system. Next, enter the MFLINK control statement on NOS. Finally, enter the permanent file server directives for NOS/VE, the remote system. These directives must include a NOS/VE login and the SCL SEND_FILE command. The SEND_FILE command sends a file from a NOS/VE system to another system. You use the SEND_FILE command to send a file from your remote NOS/VE system on one mainframe to the local NOS system on another mainframe.

The SCL System Interface Usage manual fully describes SEND_FILE. A commonly used format for SEND_FILE follows:

```
SEND_FILE  or
SENF
    FILE=file
```

The FILE or F parameter specifies the name of the file on the remote system that is to be sent to your local system. This file must be a permanent file residing on the remote system.

The following example transfers permanent file $USER.PERM_NOS on the remote NOS/VE system to temporary file LOCOPY on the NOS local system.

```
MFLINK,LOCOPY,ST=SUN
```

LOCOPY names the NOS temporary file to receive the file from the remote system. The remote NOS/VE system has logical identifier SUN.

```
*LOGIN USER=USERA  PW=BLUE  FN=SKY
```

Log in for the remote NOS/VE system with user name USERA, password BLUE, and family name SKY.

The * is the NOS prompt for the permanent file server directives.

```
*COPF  I=.USERB.STARS O=$USER.PERM_NOS
```

Creates permanent file $USER.PERM_NOS from USERB's permanent file STARS.

```
*SEND_FILE  FILE=$USER.PERM_NOS
```

Sends $USER.PERM_NOS from the remote NOS/VE system to temporary file LOCOPY in the local NOS system.

```
* (Carriage return)
```

Indicates end of the permanent file server directives.

# File Management Utility (FMU)

The File Management Utility (FMU) of NOS/VE provides comprehensive capabilities for converting and transferring files from NOS to NOS/VE, and vice versa. FMU can also convert NOS/VE files from one NOS/VE format to another.

## FMU Overview

The following questions about FMU and answers provide a quick introduction to the utility, its functions and capabilities.

### What Can FMU Do?

FMU can perform the following operations for you:

Copy files.

Convert files from one file organization to another.

Migrate files between the NOS and NOS/VE sides of a dual state system.

Migrate NOS tape files to NOS/VE, either through an interstate connection or from a NOS/VE tape drive.

Select records from the input file to have specified data conversions performed.

Reformat records by performing the following operations: reorder data fields, convert from one data type to another, insert literals, suppress zeros or blanks, pack or expand data, and truncate data.

Place a sequence number in each record.

Format a file for printing.

### What Files Can FMU Handle?

FMU can migrate sequential, indexed sequential, and direct access files.

If you need to migrate files of other organizations, consider the following suggestions:

- Remember that you can process in dual state on your CYBER. That is, you can use either NOS or NOS/VE. You probably do not have to migrate all applications immediately because both systems are available. Migrate the easiest ones first.

- NOS files with actual key file organization (FO=AK in the FILE command) can be converted with the NOS utility FORM to a NOS indexed sequential file. The file can be migrated as an indexed sequential file. See the FORM reference manual for information about the utility.

- NOS word addressable file.

  If the file has fixed length records (record type F) and each record begins on a word boundary, the file can be migrated as a sequential file with FMU. You would specify FO=SQ (FILE_ORGANIZATION = SEQUENTIAL) for the description of the file.

  If the word addressable file does not meet the requirements above, it could be converted with FORM to a NOS sequential file and migrated as a sequential file.

- NOS/VE byte addressable files are not specifically handled by FMU; however, they can be processed as sequential files. (See the SCL Advanced File Management Usage manual for details.)

You can also use FMU to migrate NOS tape files on NOS/VE tape drives to NOS/VE. Migrating tape files on NOS/VE tape drives is nearly identical to migrating other files, with the following exceptions:

- Before calling FMU, you must create a tape request. You do this with the CREATE_170_REQUEST command.

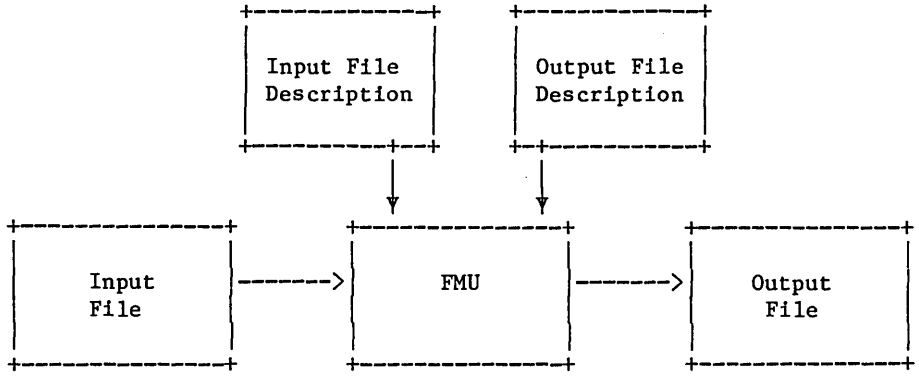
- You specify the attributes of the file to be migrated on the CREATE_170_REQUEST command instead of a FILE statement.

- Also before calling FMU, you must specify certain file attributes for the output NOS/VE file.

The FMU techniques described in the rest of this chapter apply to both disk and tape files. Refer to Migrating Tape Files in chapter 12 for information specific to tape files.

### How Does FMU Work?

FMU works by your providing file descriptions of the file to be converted (called the input file) and file descriptions of the converted file (called the output file). FMU uses your descriptions in converting and writing the new file. The process is shown in the following diagram:



In some simple situations (NOS/VE file to NOS/VE file conversion), the only file descriptions required are file attributes. This is the situation of the FMU command copy. FMU obtains information about the input file from the file information table associated with the file. Then FMU either uses default values or assumes information from the input file in writing the output file. You can also specify information for the output file in a SET_FILE_ATTRIBUTES (SETFA) command.

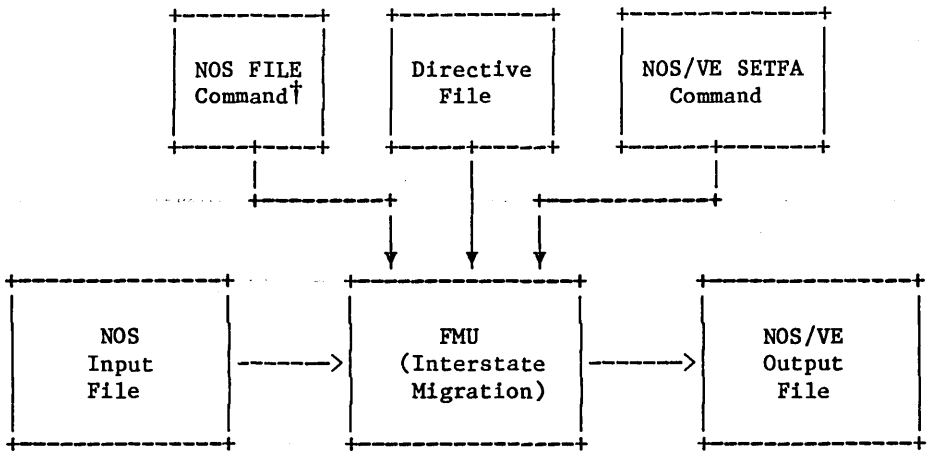
Sometimes FMU does not provide the correct values for the record type and block type of the output file. To ensure the correct values, include the following command before calling FMU:

```
SET_FILE_ATTRIBUTES  FILE=output file ..
    RECORD_TYPE=keyword ..
    BLOCK_TYPE=keyword
```

For information about which values to assign to RECORD_TYPE and BLOCK_TYPE, see the CYBIL File Interface Usage manual or the appropriate programming language usage manual.

### How Does FMU Handle NOS Files?

For migrating files between NOS and NOS/VE, FMU requires directives describing the files to reside in a directive file. These conversions also require that the NOS/VE file attributes be described with a SET_FILE_ATTRIBUTES command and that the NOS file attributes be described with a NOS FILE command or, for NOS tape files on a NOS/VE drive, a CREATE_170_REQUEST command. This situation is shown in the following diagram.




†For NOS tape files on a NOS/VE tape drive, use a CREATE_170_REQUEST command instead of a FILE command.

See the discussion of Interstate Connection Commands in chapter 4 for more information.

### How Can I Use This Discussion of FMU?

The discussion of FMU demonstrates using FMU and indicates its usefulness to you. The remainder of the discussion is divided into the following topics:

FMU command copy

FMU command copy format

FMU directive file

FMU command format with directive file

Example migrating an indexed sequential file containing character data from NOS to NOS/VE

Example migrating an indexed sequential file containing character data from NOS/VE to NOS

Example migrating a file containing binary data from NOS to NOS/VE

To construct anything except simple FMU tasks, you should seek additional information in the SCL Advanced File Management Usage manual.

## FMU Command Copy

The FMU command copy capability performs simple file copying from one NOS/VE format to another. (The command copy does not apply to NOS files.) The input and output files are specified in parameters in the FMU command. For example, assume that you wish to copy an indexed sequential file called ISFILE to a sequential file called SQFILE. The command appears as follows (file LIST is designated as the list file):

```
/fmu  input=isfile  output=sqfile  l=list
```

On a command copy, the file organization assumed for the output file is sequential. Record length and type, access mode, and most other attribute values are determined by corresponding values in the input file. If you want to specify other attribute values, you need to specify a SET_FILE_ATTRIBUTES (SETFA) command to describe the output file to FMU.

FMU knows the characteristics of the input file because a file information table is always associated with an established file.

The FMU command copy is useful if you wish to list an indexed sequential file at your terminal. Currently, the COPY_FILE command cannot handle an indexed sequential file, but it can handle the sequential file to which the indexed sequential file is copied. Therefore, you can indirectly list your indexed sequential file ISFILE by copying SQFILE:

```
/copy_file  input=sqfile  <--- Lists the contents of file SQFILE.
```

The next example shows using the FMU command copy to create an indexed sequential file from a sequential file.

### Using the FMU Command Copy to Create an Indexed Sequential File

Sequential file ANIMALS contains information about animals and their associated habitats. The file with the field to be designated as the key field in the indexed sequential file appears as follows:

```
1          11         21         31   36   41   46  50  <--- Position
|-blanks---|----------|-key------|----|----|----|---|
           MAMMAL     LION       1    LAND
           BIRD       DUCK       3    AIR  LAND WATER
           MAMMAL     SEAL       2    LAND WATER
           FISH       SHARK      1    WATER
           MAMMAL     WHALE      1    WATER
           BIRD       PENGUIN    2    LAND WATER
```

The following file attributes are needed to describe the file as an indexed sequential file to FMU:

| | |
|---|---|
| BT=SS | BLOCK_TYPE is SS (for system specified) as opposed to US (user specified). |
| FO=IS | FILE_ORGANIZATION is indexed sequential (IS). |
| KL=10 | KEY_LENGTH is 10 characters. |
| KP=20 | KEY_POSITION is at position 20 (KP + 1 defines the location of the first character of the key.) |
| KT=UNCOLLATED | KEY_TYPE is uncollated, which means the 7-bit ASCII sequence applies when key values are compared. |
| MAXRL=50 | MAXIMUM_RECORD_LENGTH is 50 characters. |
| MINRL=35 | MINIMUM_RECORD_LENGTH is 35 characters. |
| RT=U | RECORD_TYPE is U (for undefined) as opposed to F (fixed). Record type could have been V (for variable). |

These attributes are specified in a SET_FILE_ATTRIBUTES (SETFA) command for file NEWIS (the indexed sequential file being created from sequential file ANIMALS). In this example, both files are local files. The command stream is:

```
/set_file_attributes  file=newis ..
.. /block_type=system_specified ..
.. /file_organization=indexed_sequential ..
.. /key_position=20 ..
.. /key_length=10 ..
.. /key_type=uncollated ..
.. /maximum_record_length=50 ..
.. /minimum_record_length=35 ..
.. /record_type=undefined
/fmu  input=animals ..  <---------- Creates file NEWIS
.. /output=newis ..
.. /l=list
```

For more information about file attributes, see the File Attributes discussion in chapter 10.

### FMU Command Copy Format

```
FMU I=infile O=outfile L=listfile ED=value STATUS=variable
```

FMU can be spelled out as FILE_MANAGEMENT_UTILITY.

Parameters: All parameters are optional, but for this discussion, assume that you specify parameters I and O.

I — (INPUT) File reference for the file being copied.

O — (OUTPUT) File reference for the file being written.

L — (LIST) File reference for the list file. Default is $LIST, which is connected to $NULL for interactive use. That means that the listing disappears unless you either designate a list file or connect standard file $LIST to a real file. (See the discussion about File Connection Commands in chapter 4, Common NOS/VE Commands).

ED — (ERROR_DISPOSITION) Specifies whether the FMU run is to be aborted if the output file is aborted (closed before the end of the run because of an error). The values are:

```
ED=A     ABORT (Default)
ED=NA    NO_ABORT
```

STATUS — Status variable; see the discussion of the STATUS parameter that appears in chapter 4, Common NOS/VE Commands.

## FMU Directive File

The directive file provides FMU with information about the input and output files. When you migrate a file from NOS to NOS/VE the NOS file is the input file and the NOS/VE file is the output file; therefore, you must specify directives to describe files being migrated. The following directives are usually specified in migration situations:

```
SET_INPUT_ATTRIBUTES (SETIA)

SET_OUTPUT_ATTRIBUTES (SETOA)

CREATE_OUTPUT_RECORD (CREOR)
```

More directives are available; however, only these are used in the examples in this guide.

## SET_INPUT_ATTRIBUTES Directive

The SET_INPUT_ATTRIBUTES (SETIA) directive describes the input file. For NOS to NOS/VE file conversion, you need to specify the following parameters:

F — (FILE) Local file name of the file containing input records.

MF — (MACHINE_FORMAT) Specified as follows:

```
MF=C170    NOS file

MF=C180    NOS/VE file (default)
```

The parameters SFP (STARTING_FILE_POSITION) and MFU (MAXIMUM_FILE_UNITS) can also be specified to designate records in the input file. See the SCL Advanced File Management Usage manual for more information.

## SET_OUTPUT_ATTRIBUTES Directive

The SET_OUTPUT_ATTRIBUTES (SETOA) directive describes the output file. For NOS to NOS/VE file conversion, you would typically use the following parameters:

F (FILE) Local file name of the file containing input records

MF (MACHINE_FORMAT) Specified as follows:

MF=C170 NOS file

MF=C180 NOS/VE file (default)

ED (ERROR_DISPOSITION) Possible specifications are:

ED=A ABORT (default).

ED=NA NO_ABORT. This specification keeps the task executing if an error occurs.

ERF (EXCEPTION_RECORD_FILE) Specifies a file to receive records causing errors during migration. Default is $NULL (no exception file produced). Note that if ERROR_DISPOSITION=ABORT is also specified, at most one record (the record that caused the abort) is written to this file.

CED (CONVERSION_ERROR_DISPOSITION) Specifies whether or not recovery is to be attempted for conversion errors caused by unrecognizable data in the source field. Options are:

CED=A ABORT. Default. No recovery is attempted.

CED=R RECOVER. FMU attempts to recover by assuming one of the following values for the source field:

Zero for a numeric field.

Spaces for a character (A) field.

False value for a logical field.

The parameters DS (DUPLICATE_SPECIFICATION), PD (PARTITION_DISPOSITION), and MFU (MAXIMUM_FILE_UNITS) can also be specified to designate records in the output file. See the SCL Advanced File Management Usage manual for descriptions of these parameters.

## CREATE_OUTPUT_RECORD Directive

The CREATE_OUTPUT_RECORD (CREOR) directive specifies exactly how you want your output record to be formatted and the logical steps necessary to perform the reformatting. This directive is required for FMU to perform conversions of binary files. The directive can specify complex record reformatting; this guide, however, shows only simple, direct data conversions.

The discussion of the CREATE_OUTPUT_RECORD directive is divided into topics to indicate how to use the CREATE_OUTPUT_RECORD directive in migration situations. The topics are:

Assignment Statements and Field Descriptors

Assignment Statement for a Double Precision Item

CREATE_OUTPUT_RECORD (CREOR) Format

FMU Data Types for FORTRAN and COBOL

Defaults for the FMU Data Types

Converting character data

For an example of using CREATE_OUTPUT_RECORD when executing FMU, see the example of converting a binary file shown later in this chapter.

### Sample Binary File Description With CREATE_OUTPUT_RECORD

The descriptions of records in the CREATE_OUTPUT_RECORD directive allow you to specify the conversions required to migrate your file. To see how this works, assume you have a binary file on NOS in which the records have the fields shown in table 11-1.

Table 11-1. Sample NOS Binary Record

| Number of Items | FORTRAN Data Item | COBOL Data Item | FMU Data Type | |
|---|---|---|---|---|
| 3 | REAL | COMPUTATIONAL-2 (NOS) | F | for floating point |
| 3 | INTEGER | COMPUTATIONAL-1 (NOS) | I | for integer |
| 1 | CHARACTER*10 | PIC X(10) or PIC A(10) | A | for alphabetic (represents string) |
| 1 | Logical | Not applicable | L | for logical |

In characters, the record appears as follows (to FMU, the record has 10-byte binary fields):

```
--RECORD 1-----------------------------------------------------+--RECORD 2---
32.654 | 165.41 | -0.0013 |  54 |  9 | 125 | ABCDEFGHIJ | TRUE | ...
-------------------------------------------------------------+-------------
 F       F         F       I     I    I      A[ ,10]     L    FMU record description
```

The CREATE_OUTPUT_RECORD directive to convert this NOS file to the equivalent NOS/VE format as file NVEFILE is:

```
create_output_record  file=nvefile
    F; F; F; I; I; I; A[ ,10]; L
create_output_record_end
```

This conversion is straight forward. The first data item in the NOS record is converted to the equivalent NOS/VE data item and stored in the NOS/VE record. The second item is converted and stored, and so forth. The directive processing is repeated for each record.

The data type specifications F, I, A, and L are FMU assignment statements indicating a conversion of one data item from one format to another. These are the simplest form of FMU assignment statements and are called single descriptor assignment statements.

### Assignment Statements and Field Descriptors

To FMU, each assignment statement represents a field description in the input and output records. In the sample CREOR directive for file NVEFILE (shown in a preceding paragraph), the field descriptions are single field descriptors. The full assignment statements equivalent to the single field descriptors are as follows:

```
F          F[ ,8]  = F[ ,10]
I          I[ ,8]  = I[ ,10]
A[,10]     A[ ,10] = A[ ,10]
L          L[ ,8]  = L[ ,10]
             |          |
             |          Describes the data field in the input file, the
             |          NOS file in this example.
             |
             Describes the data field in the output file, the NOS/VE file
             for this example.
```

A further breakdown of the field descriptors is as follows:

```
F[ ,8] =  F[ ,10]
|| |      || |
|| |      || Length in bytes of the field in the input record (6-bit bytes for NOS).
|| |      |
|| |      Default byte position of the field in the input record.
|| |
|| FMU data type of the field in the input record (the NOS file).
||
| Length in bytes of the field in the output record (8-bit bytes for NOS/VE).
|
Default beginning byte position of the field in the output record.
|
FMU data type of the field in the output record (the NOS/VE file).
```

For the default byte position, FMU uses the position following the previously described item. With default specifications, FMU works sequentially through both the input and output files. In descriptor F[5,8], 5 specifies position.

### Assignment Statement for a Double Precision Item

The complete assignment statement is required for converting double precision items because FMU assumes single precision conversion if just the F descriptor is specified. The assignment statements required for converting double precision items are:

```
From NOS to NOS/VE      F[ ,16] = F[ ,20]

From NOS/VE to NOS      F[ ,20] = F[ ,16]
                          |          |
                          |          Describes the field for the input record.
                          |
                          Describes the data field in the output record.
```

Notice that by switching the field descriptor from one side of the equals sign (=) to the other, you change the direction of the conversion: from NOS to NOS/VE or from NOS/VE to NOS. (The left descriptor is always for the output file; the right descriptor is always for the input file.)

### CREATE_OUTPUT_RECORD (CREOR) Format

```
CREOR F=newlfn RPV=op;
    statement-list;
CREOREND
```

Parameters:

F — (FILE) specifies the local file name of the output file (the new file being created).

RPV — (RECORD_PRESET_VALUE) Specifies that the input record fields not referenced in the CREATE_OUTPUT_RECORD statement-list can be set in the output record as follows:

- NP — NO_PRESET (default value)
- CB — CHARACTER_BLANK
- CZ — CHARACTER_ZERO
- BZ — BINARY_ZERO
- IR — INPUT_RECORD; that is the data in the output record is to be the same as the corresponding data in the input record, unless altered by assignment statements.

statement-list — Can consist of assignment statements as shown in the example below:

```
F; F; F; I; I; I; A[ ,10]; L
```

A semicolon must separate statements that appear on the same line.

The assignment statements are discussed on preceding pages. See subsequent pages for tables that list FMU data type notation used in migrating COBOL or FORTRAN generated files.

CREOREND — (CREATE_OUTPUT_RECORD_END) Terminates the CREATE_OUTPUT_RECORD directive.

For more information about the formats required for FMU, see the SCL Advanced File Management Usage manual.

### FMU Data Types for FORTRAN and COBOL

The FMU descriptors that are used in the CREATE_OUTPUT_RECORD directive can describe data items produced by FORTRAN and COBOL applications. Table 11-2 specifies the FMU descriptor for corresponding FORTRAN or COBOL data items.

Table 11-2. FMU Data Types for FORTRAN and COBOL

| FMU Data Type | NOS | | NOS/VE | |
|---|---|---|---|---|
| | FORTRAN 5 | COBOL 5 | FORTRAN | COBOL |
| A | Character | PIC X | Character | PIC X |
| B | Any | Any | Any | Any |
| F.10/8 | Real | COMP-2 | Real | COMP-1 |
| F.20/16 | Double precision | N/A | Double precision | COMP-2 |
| G | Character | N/A | Character | N/A |
| H | N/A | Any PIC S99... | N/A | Any PIC S99... |
| I | Integer | COMP-1 or COMP-4 | Integer | COMP PIC S9... |
| J | N/A | Any PIC S(9) SIGN IS LEADING | N/A | Any PIC S(9) SIGN IS LEADING |
| L | Logical | N/A | Logical | N/A |
| N | Character | Any PIC ...ZZZ9 | Character | Any PIC ...ZZZ9 |
| P | N/A | N/A | N/A | COMP-3 PIC S9. |
| Q | N/A | N/A | N/A | COMP-3 PIC 9... |
| U | N/A | PIC 99... | N/A | PIC 99... UNSIGNED UNPACKED DECIMAL |
| Y | Character | Any PIC S(n) SIGN IS TRAILING SEPARATE | Character | Any PIC S(n) SIGN IS TRAILING SEPARATE |
| Z | Character | Any PIC S999... SIGN IS LEADING SEPARATE | Character | Any PIC S999... SIGN IS LEADING SEPARATE |

Note: F.10/8 and F.20/16 indicate the ratio of bytes in converting NOS floating point to NOS/VE. Use F (alone) to convert single precision. (The assignment statement for double precision is shown earlier in this chapter.)

Each FMU data type has an associated default length. Table 11-3 gives the default length in bytes for FMU data types. In a migration situation, when just the data type designator (for example, I or A) is specified, FMU performs conversion according to the default values.

Table 11-3. Default Lengths for FMU Data Types

| Data Type | Description | NOS/VE Default | NOS Default |
|---|---|---|---|
| A | Alphanumeric character string | 1 | 1 |
| Z | Integer character string with leading zeros and leading sign | 19 | 19 |
| Y | Integer character string with leading zeros and trailing sign | 19 | 19 |
| N | (NORMAL) integer character string with leading blanks | 19 | 19 |
| G | (GENERAL) floating point character string | 22 | 22 |
| H | Trailing sign combined Hollerith; also called trailing overpunch | 18 | 18 |
| J | Leading sign combined Hollerith; also called leading overpunch | 18 | 18 |
| P | Signed packed decimal | 19 | not applicable |
| Q | Unsigned packed decimal | 19 | not applicable |
| U | Unsigned unpacked decimal | 18 | not applicable |
| L | Logical | 8 | 10 |
| I | Signed integer | 8 | 10 |
| F | Floating-point, for both single and double precision (single precision is the default length) | 8 | 10 |
| B | Unsigned binary | 1 bit | 1 bit |

Length is given in bytes unless otherwise noted. (NOS/VE uses 8-bit bytes. NOS uses 6-bit bytes.)

### Character Data Conversion

FMU supports conversion between the NOS/VE ASCII character set and the following NOS character sets:

63-character display code

64-character display code

6/12 ASCII

8/12 ASCII

The following restrictions apply to files that use the 6/12 or 8/12 ASCII set:

A SET_PRINT_ATTRIBUTES (SETPA) FMU directive cannot be specified for an output file whose character set is 6/12 or 8/12 ASCII.

The $INPUT_STRING_POSITION ($ISP) function cannot be used to test an input file whose character set is 6/12 or 8/12 ASCII.

The SET_PRINT_ATTRIBUTES directive and $INPUT_STRING_POSITION function are described in the SCL Advanced File Management Usage manual.

For a NOS/VE file, the character set used by the file is specified by the INTERNAL_CODE (IC) parameter on the SET_FILE_ATTRIBUTES (SETFA) command. The default and only valid option for the INTERNAL_CODE parameter is ASCII.

For a NOS file, the character set used by the file is specified by the IC parameter on the FILE command. The IC parameter can have the following values:

D63 63-character set

D64 64-character set

A612 6/12 bit ASCII

A812 8/12 bit ASCII

The default, either D63 or D64, is site-dependent.

The CHARACTER_CONVERSION parameter (on the SET_FILE_ATTRIBUTES command) and the CC parameter (on the FILE control statement) are not needed for character conversion.

The length of a character data field (FMU type A) is specified as a number of 8-bit bytes (NOS/VE) or 6-bit bytes (NOS). FMU performs a left-to-right byte move until the destination field is full.

If the source field is longer than the destination field, the data item is truncated on the right. If the source field is shorter than the destination field, the destination field is padded on the right with blanks. In addition, the following applies to 6/12 and 8/12 ASCII conversion.

For 8/12 ASCII fields, the actual number of characters in the field is one half the length of the field. Thus, for example, the descriptor

A[,20]

describes a 20-byte field that contains 10 characters.

Odd-length 8/12 ASCII fields cause an execution error.

For 6/12 ASCII fields, the number of characters in a field of length N is variable, ranging from N for an all-uppercase field to N/2 for an all-lowercase field.

If either of the following conditions occur when converting a 6/12 ASCII field, FMU issues an execution diagnostic:

The last character of the source field is 12 bits long and starts in the last 6 bits of the field.

The destination field is not long enough to hold the entire string assigned to it, and the string would be truncated in the middle of a 12-bit character.

Both of these errors can be suppressed by specifying ERROR_DISPOSITION=NO_ABORT on the SET_OUTPUT_ATTRIBUTES directive. They then become nonfatal conversion errors, and in both cases, spaces are assigned to the field that was too short to contain the character assigned to it.

If a file to be migrated consists entirely of character data, you do not need to specify a CREATE_OUTPUT_RECORD (CREOR) directive to describe the input and output record formats. If you omit the CREATE_OUTPUT_RECORD directive, FMU automatically performs a character conversion on all data in the file. This method of character conversion results in faster FMU execution than if a CREATE_OUTPUT_RECORD directive were specified.

## Migrating COBOL Records

COBOL inserts slack bytes in data records to align binary data on word boundaries and to cause repeating groups to align on common byte offsets. (That is, each occurrence of a repeating group must begin at the same byte offset within a word.) You must take these slack bytes into account when specifying the data field starting byte positions in the FMU field descriptors.

The best way to determine the starting byte position of data fields in COBOL records is to examine the data map generated as part of the COBOL compiler output listing (LO=M option on the COBOL5 command.) For each data item, this map shows the relative address of the word containing the data item and the byte offset within the word at which the data item begins. You can use this information to construct the FMU field descriptors necessary to convert the data records.

The following is an example of migrating a COBOL record that contains slack bytes. The record description in the COBOL source program is as follows:

```
01  OUT-REC.
      02  ITEM-1 PIC X(14).
      02  ITEM-2 OCCURS 3 TIMES.
            03  ITEM-2A PIC XX.
            03  ITEM-2B PIC 999V99 COMP-1.
            03  ITEM-2C PIC X(8).
```

COBOL will insert slack bytes to align the COMP-1 item on a word boundary, and to align each occurrence of the repeated group ITEM-2 on a common byte offset within a word.

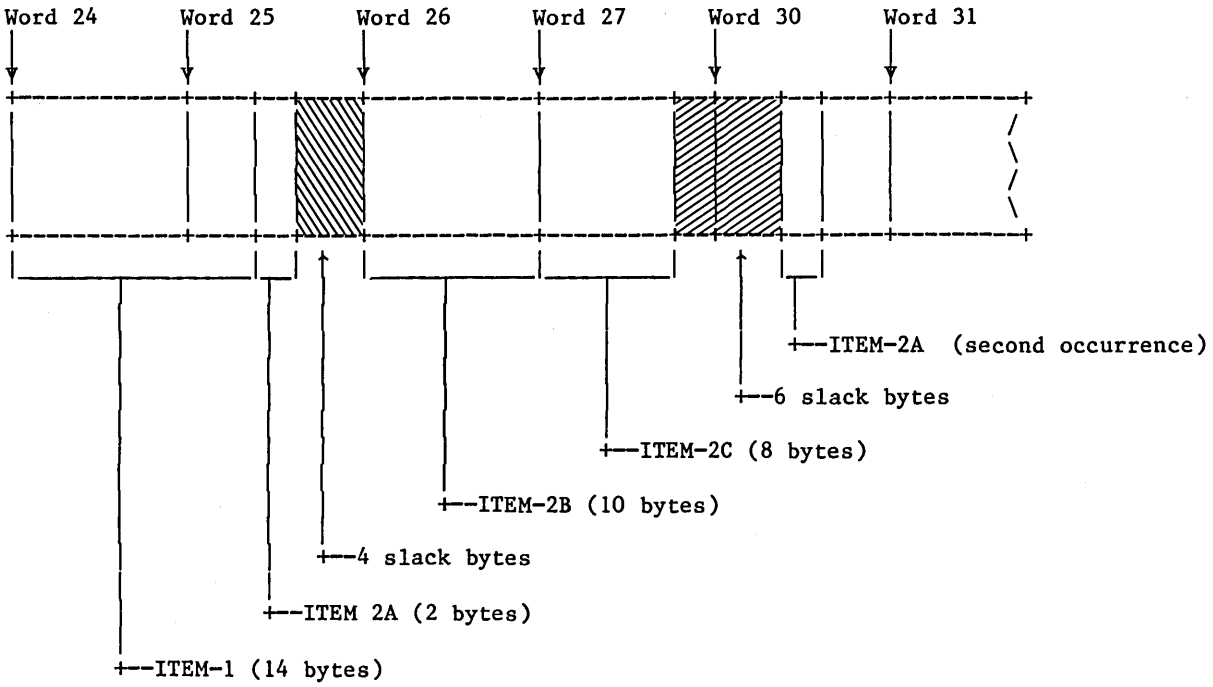
The data map from the COBOL output listing is as follows:

```
01  OUT-REC              000024/00 SZ=104
02  ITEM-1               000024/00    14
02  ITEM-2               000025/04    30
03  ITEM-2A              000025/04    2
03  ITEM-2B              000026/00    10
03  ITEM-2C              000027/00    8
```

The third column of the data map contains entries of the form nn/mm, where nn is the relative word address (octal) of the data item, and mm is the byte (octal) within the word where the item begins. The fourth column gives the length (decimal bytes) of each data item.

For example, ITEM-1 begins in byte 0 of word 24 of the program and is 14 bytes long; ITEM-2 (a repeated group) begins in byte 4 of word 25 and is a total of 30 bytes long. The common byte offset for each occurrence of group ITEM-2 is 4 bytes.

The following diagram shows the format of the data record in memory. The shaded areas indicate slack bytes inserted by COBOL.




The four slack bytes following ITEM-2A align ITEM-2B (a COMP-1 item) on a word boundary. The six slack bytes following ITEM-2C cause the second occurrence of ITEM-2 to align on the common byte offset of 4.

Given that one NOS word contains 10 bytes, the Word/Offset values from the data map can be converted to starting byte positions relative to the beginning of the record. The starting byte positions of the data items in the example are as follows:

| Item | Word/Offset (from data map) | Length | Starting Byte Position Within Record |
|---|---|---|---|
| ITEM-1 | 24/00 | 14 | 1 |
| ITEM-2A | 25/04 | 2 | 15 |
| ITEM-2B | 26/00 | 10 | 21 |
| ITEM-2C | 27/00 | 8 | 31 |

Since the common byte offset of repeated group ITEM-2 is 4, the next occurrence of the group begins in byte 4 of word 30 (octal). The starting byte positions of the second occurrence of ITEM-2 are as follows:

| | |
|---|---|
| ITEM-2A | 45 |
| ITEM-2B | 51 |
| ITEM-2C | 61 |

The starting byte positions of the third occurrence of ITEM-2 are as follows:

| | |
|---|---|
| ITEM-2A | 75 |
| ITEM-2B | 81 |
| ITEM-2C | 91 |

The FMU field descriptors needed to migrate the record can now be written. A descriptors are used to migrate the PIC XX... fields, and I descriptors are used to migrate the COMP-1 fields. The complete CREATE_OUTPUT_RECORD directive for this example is as follows:

```
CREATE_OUTPUT_RECORD FILE=filename

    A[,14]=A[1,14]  "Migrate ITEM-1"

    A[,2]=A[15,2]; I[,8]=I[21,10]; A[,8]=A[31,8]  "Migrate ITEM-2"

    A[,2]=A[45,2]; I[,8]=I[51,10]; A[,8]=A[61,8]  "Migrate ITEM-2"

    A[,2]=A[75,2]; I[,8]=I[81,10]; A[,8]=A[91,8]  "Migrate ITEM-2"

CREATE_OUTPUT_RECORD_END
```

## FMU Command Format With Directive File

With this format, the input and output files are specified in the directive file. The DIRECTIVES parameter of the FMU command identifies the directive file. All parameters are optional, but for this discussion assume that you specify the DIRECTIVES parameter.

Format:

```
FMU DIR=directive-file L=listfile ED=value STATUS=variable
```

FMU can be spelled out as FILE_MANAGEMENT_UTILITY.

Parameters:

DIR — (DIRECTIVES) File reference to identify the file containing input directives. There is no default value.

L — (LIST) File reference for the list file. Default is $LIST, which is connected to $NULL for interactive use. That means that the listing disappears unless you either designate a list file or connect standard file $LIST to a real file. (For more information, see the File Connections discussion in chapter 4, Common NOS/VE Commands.)

ED — (ERROR_DISPOSITION) Specifies whether the FMU run is to be aborted if the output file is aborted (closed before the end of the run because of an error). The values are:

```
ED=A     ABORT (Default)
ED=NA    NO_ABORT
```

STATUS — Status variable; see the discussion of the STATUS parameter that appears in chapter 4, Common NOS/VE Commands.

## Example Migrating a Simple NOS Indexed Sequential File to NOS/VE

This example converts a NOS indexed sequential (IS) file containing only character data to a NOS/VE indexed sequential file. The file being converted is the animals and habitats file used in preceding example and shown again below.

```
1         11         21        31   36   41   46  50   <--- Position
|-blanks--|----------|-key------|----|----|----|---|
          MAMMAL     LION      1    LAND
          BIRD       DUCK      3    AIR  LAND WATER
          MAMMAL     SEAL      2    LAND WATER
          FISH       SHARK     1    WATER
          MAMMAL     WHALE     1    WATER
          BIRD       PENGUIN   2    LAND WATER
```

### Describing the Sample Files

FMU requires descriptions of both the input file (ISFILE) and output file (ISNVE).

#### Describing the Input File

The description for the input file must identify the file and provide some specific information about the kind of the file. Detailed record descriptions are not required because these can be determined by the file interface once the file is recognized. Input file descriptions require the NOS FILE command and the FMU SET_INPUT_ATTRIBUTES directive.

The NOS FILE command provides information as follows:

| | |
|---|---|
| ISFILE | Local file name. |
| FO=IS | File organization is indexed sequential. |
| ORG=NEW | The file is an extended indexed sequential file. NEW is the default value for NOS 2; however, specifying ORG=NEW is safe and could avoid a problem. |

The SETIA (SET_INPUT_ATTRIBUTES) directive specifies information as follows:

| | |
|---|---|
| ISFILE | Identifies the file |
| MACHINE_FORMAT=C170 | Designates a NOS file |

### Describing the Output File

The description for the output file must identify the file and provide specific information about the record and primary key. Output file descriptions require the SET_FILE_ATTRIBUTES command and the SET_OUTPUT_ATTRIBUTES directive. Because the file contains only character data, the CREATE_OUTPUT_RECORD directive is not required.

The NOS/VE SET_FILE_ATTRIBUTES command provides information as follows:

| | |
|---|---|
| file=isnve | Local file name for the file in the example. |
| block_type=system_specified | Block type is system specified, as opposed to user specified (US). |
| data_padding=80 | 80 percent data padding. |
| index_padding=10 | 10 percent index padding. |
| embedded_key=yes | Embedded key. |
| file_organization=indexed_sequential | File organization is indexed sequential. |
| key_length=10 | Key length is 10 characters. |
| key_type=uncollated | Key type is uncollated, meaning that the 7-bit ASCII sequence applies to comparisons of key values. |
| maximum_record_length=50 | Maximum record length is 50 characters. |
| minimum_record_length=35 | Minimum record length is 35 characters. |
| record_type=undefined | Record type is undefined, as opposed to fixed (F). This could have been V (for variable). |

The SET_OUTPUT_ATTRIBUTES directive specifies information as follows:

| | |
|---|---|
| file=isnve | Identifies the file. |
| machine_format=C180 | This is not specified but is the default value. |
| error_disposition=no_abort | Specifies no abort if an error occurs on the output file. |

### Commands in the Job Stream

The task is set up as a procedure called ISCONVT. The commands PROC and PROCEND begin and end the procedure, respectively.

The FMU command is included twice: first to perform the NOS to NOS/VE conversion; second, to perform a command copy to convert the NOS/VE indexed sequential file to a sequential file for copying to the terminal. The COPY_FILE command following the second FMU command produces the file listing in the example. Other commands requiring further explanation are:

CREFC (CREATE_FILE_CONNECTION) Connects standard file $ECHO to file OUTPUT so that the executing commands are copied to the terminal as they execute.

CREIC (CREATE_INTERSTATE_CONNECTION) Allows executing NOS commands. See the Interstate Connection discussion in chapter 4, Common NOS/VE Commands, for a description of this facility.

COLT (COLLECT_TEXT) Creates the directive file DIRFILE.

### Procedure ISCONVT and Execution Output

All the commands for converting ISFILE to ISNVE are contained in procedure ISCONVT. For the procedure to execute, file ISFILE must exist as a permanent direct access file on NOS. If the procedure is developed on NOS and stored as file ISCONVT, it can be transferred to NOS/VE with the GETF command (get_file to=isconvt) and executed by specifying the file name: ISCONVT. The commands in the procedure ISCONVT are shown in figure 11-1.

```
Command                                                  Comment

proc isconvt  <--------------------------------- Identifies the procedure.
setmm full  <----------------------------------- SET_MESSAGE_MODE to full.
crefc $echo output  <--------------------------- Displays procedure commands.
creic  <---------------------------------------- Connects with NOS.
exeic 'ATTACH,ISFILE.'  <----------------------- Attaches NOS file ISFILE.
exeic 'FILE,ISFILE,FO=IS,ORG=NEW.'  <----------- Describes NOS file.
cref $user.isnve  <----------------------------- Creates permanent file.
colt dirfile  <--------------------------------- Creates directive file.
setia isfile mf=cl70  <------------------------- Directive for input file.
setoa isnve ed=na  <---------------------------- Directive for output file.
**  <--------------------------------------------- Terminates collect text.
setfa isnve bt=ss dp=80 ip=10 ed=yes fo=is ..  <-- Describes NOS/VE file, see preceding
  kl=10 kp=20 kt=uc maxrl=50 minrl=35 ..           paragraphs for the explanation.
  rt=u
fmu dir=dirfile l=fmuout  <--------------------- Calls FMU.
quit  <----------------------------------------- Terminates NOS connection.
fmu i=isnve o=see_is_file  <-------------------- FMU command copy.
copf see_is_file  <----------------------------- Copies file to terminal.
delfc $echo,output  <--------------------------- Deletes connection to $ECHO.
procend isconvt  <------------------------------ Terminates procedure.
```

Figure 11-1. Procedure ISCONVT

The output from the procedure execution is shown in figure 11-2.

```
CI creic
CI exeic 'ATTACH,ISFILE.'
CI exeic 'FILE,ISFILE,FO=IS,ORG=NEW.'
CI cref $user.isnve
CI colt dirfile
CI setfa isnve bt=ss dp=80 ip=10 ek=yes fo=is kl=10 ip=20 kt=uc maxrl=50 minrl=35 rt=u
CI fmu dir=dirfile l=fmuout
CI quit
CI fmu i=isnve o=see_is_file
CI copf see_is_file
          BIRD      DUCK       3    AIR  LAND WATER
          MAMMAL    LION       1    LAND
          BIRD      PENGUIN    2    LAND WATER
          MAMMAL    SEAL       2    LAND WATER
          FISH      SHARK      1    WATER
          MAMMAL    WHALE      1    WATER
CI delfc $echo,output
```

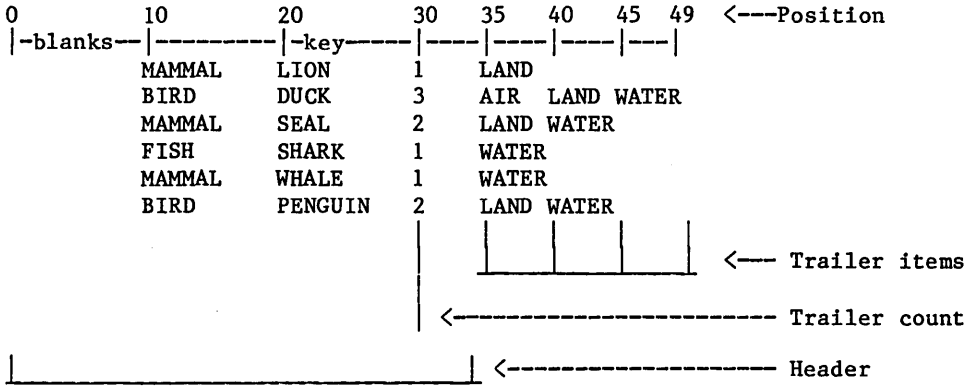
Figure 11-2. Execution Output of Procedure ISCONVT

## Example Migrating a Simple NOS/VE IS File to NOS

This example converts a NOS/VE indexed sequential (IS) file containing character data to a NOS indexed sequential file. The file being converted is the same one converted in the previous example. It is being migrated back to NOS to show the reverse process.

Most of the commands are similar to those in the previous example. Reading about them should be sufficient to understand the commands in this example.

To transfer the file to NOS, the file must be described in detail. The FILE command describes records in the NOS file NEWIS, which is being created. The file is shown below to indicate data positions on NOS (start counting characters within each line from 0).




The parameters of the FILE command required to describe the file under NOS are as follows:

| | |
|---|---|
| NEWIS | Local file name of the sample file. |
| FO=IS | File organization is indexed sequential. |
| ORG=NEW | Specifies an extended indexed sequential file. This is the default value. Specifying NEW is safe and can sometimes avoid problems. |
| RT=T | Record type T (for trailer). |
| HL=35 | Header length is 35 characters (T-type records only). |
| TL=5 | Trailer length is 5 characters (T-type records only). |
| CP=30 | Trailer count beginning character position (T-type records only). Begin counting at 0. |
| CL=1 | Trailer count field length is 1 character (T-type records only). |
| MRL=50 | Maximum record length is 50 characters. |
| DP=80 | Data padding is 80 percent |
| IP=20 | Index padding is 20 percent. |
| MBL=600 | Maximum block length is 600 characters. |
| ON=NEW | OLD/NEW file is NEW. Required for a file being created. |

KT=S  Key type is symbolic (for character keys).

KL=10  Key length is 10 characters.

RKW=2  Relative key word is 2 (start counting at 0).

RKP=0  Relative key position is 0 (key begins in bit 0 of word 2).

EMK=YES  Embedded key (must be specified on NOS because the default value is NO).

EFC=3  Error file control designates that error messages and statistics are written on the error file.

For more information about the FILE command, see the CYBER Record Manager Advanced Access Methods reference manual.

The conversion job is shown as a procedure name ISBACK in figure 11-3.

```
Command                                                    Comment

proc isback  <------------------------------------------- Identifies the procedure.
setmm full  <-------------------------------------------- SET_MESSAGE_MODE to full.
creic $echo output  <------------------------------------ Displays procedure command.
creic  <------------------------------------------------- Connects with NOS.
exeic 'DEFINE,NEWIS.'  <--------------------------------- Defines output file on NOS.
exeic 'FILE,NEWIS,FO=IS,ORG=NEW,RT=T,HL=35,TL=5,CP=30.'  <-- FILE command describes NOS file.
exeic 'FILE,NEWIS,CL=1,MRL=50,DP=80,IP=20,MBL=600,ON=NEW.'
exeic 'FILE,NEWIS,KT=S,KL=10,RKW=2,RKP=0,EMK=YES,EFC=3.'
colt dirfile  <------------------------------------------ Creates directive file.
setia isnve  <------------------------------------------- Describes input NOS/VE file.
setoa newis mf=cl70 ed=na  <----------------------------- Describes output NOS file.
**  <------------------------------------------------------ Terminates collect text.
attf $user.isnve  <-------------------------------------- Attaches input file ISNVE.
fmu dir=dirfile l=fmuout  <------------------------------ Executes FMU.
exeic 'FLSTAT,NEWIS,FLSOUT.'  <-------------------------- Executes CRM FLSTAT utility.
exeic 'REPLACE,FLSOUT.'  <------------------------------- Stores FLSTAT output file.
getf flsout  <------------------------------------------- Transfers the file.
copf flsout  <------------------------------------------- Prints the file.
delfc $echo output  <------------------------------------ Deletes connection to $ECHO.
quit  <-------------------------------------------------- Terminates NOS connection.
procend isback  <---------------------------------------- Terminates procedure.
```

Figure 11-3. Procedure ISBACK

The output of this job consists of the command listing generated by the connection to $ECHO and the listing of the file FLSOUT generated by the NOS FLSTAT utility. FLSTAT prints statistics and information about the file NEWIS and is used in this task as a quick check on the FMU run. Notice that the output file for FLSOUT must be made permanent (REPLACE command) before it can be transferred to NOS/VE.

## Example of Migrating a Binary Data File With FMU From NOS to NOS/VE

This example converts a sequential NOS binary file to a NOS/VE binary file. The sample NOS binary file to migrate contains records as shown in table 11-4.

Table 11-4. Sample Binary File

| Number of Items | FORTRAN Data Item | COBOL Data Item | FMU Data Type |
|---|---|---|---|
| 5 | REAL | COMPUTATIONAL-2 (NOS) | F for floating point |
| 3 | INTEGER | COMPUTATIONAL-1 (NOS) | I for integer |
| 1 | LOGICAL | Not applicable | L for logical |

The file is created by a NOS FORTRAN 5 program, which actually creates two files: a binary file BINFILE and a file containing character data CHRDATA. File CHRDATA is listed to show the data the binary file contains.

The file creation job is a NOS CYBER Control Language (CCL) procedure.

```
.PROC,CREATE.
DEFINE,BINDATA.
DEFINE,CHRDATA.
FTN5,I=SOURCE,L=FTNOUT,REW.
LGO.
REWIND,CHRDATA.
COPY,CHRDATA.
.DATA,SOURCE
      PROGRAM WREXAMP
      REAL RNUMS(5)
      INTEGER INUMS(3)
      LOGICAL TRUTH
      DATA RNUMS /1.0,3.0,5.0,7.0,8.0/
      DATA INUMS /6,4,2/
      DATA TRUTH /.FALSE./
      OPEN (2, FILE='BINDATA', FORM='UNFORMATTED')
      OPEN (4, FILE='CHRDATA', FORM='FORMATTED')
      DO 10 I=1,10
      WRITE (2) RNUMS,TRUTH,INUMS
   10 WRITE (4, FMT='(5F7.2,L4,I7,I6,I5)') RNUMS,TRUTH,INUMS
      CLOSE (2)
      CLOSE (4)
      END
```

A listing of file CHRDATA shows the data the binary file BINFILE contains. CHRDATA appears as follows:

```
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
 1.000    3.000    5.000    7.000    8.000    F    6    4    2
```

### Sample FMU Job Migration Binfile

The FMU job to migrate file BINDATA is shown as an SCL procedure named BINCVT shown in figure 11-4. To execute this procedure on NOS/VE, create a file containing only the procedure; then specify the file name to execute the commands in the file. This example, like all the examples in this manual, assumes that the working catalog on NOS/VE is $LOCAL (by default).

```
proc bincvt  <---------------------------- Begins the SCL procedure.
setmm full  <----------------------------- Sets message mode to full.
crefc $echo echout  <--------------------- Creates a file connection to write commands to
                                           file ECHOUT as they are processed.
creic  <---------------------------------- Invokes the interstate connection.
exeic 'ATTACH,BINDATA.'  <----------------- NOS ATTACH command.
exeic 'FILE,BINDATA,MRL=90,BT=I,RT=W.'  <-- NOS FILE command describing the file being
                                           migrated.
exeic 'FILE,BINDATA,CC=NO.'  <------------- FILE command continued.  CC=NO is required for
                                           a binary conversion.
cref $user.bindata_nve  <------------------ Creates an empty file in your master catalog on
                                           NOS/VE and attaches the file as a local file.
setfa $user.bindata_nve bt=ss rt=v  <------ Sets the BLOCK_TYPE and RECORD_TYPE attributes
                                           for the new file.
colt dirfile  <---------------------------- Begins creating file DIRFILE to contain
                                           directives for FMU.
setia bindata mf=c170  <------------------- FMU directive for the input file, which is NOS
                                           file BINDATA.
setoa bindata_nve mf=c180 ed=na  <--------- FMU directive for the output file, which is
                                           BINDATA_NVE on NOS/VE.
creor bindata_nve rpv=no_preset  <--------- FMU directive describing the data conversions.
     F; F; F; F; F; L; I; I; I             (CREOR is described earlier in this chapter.)
creorend
**  <---------------------------------------- Ends text for the directive file.
fmu dir=dirfile l=fmuout  <---------------- Calls FMU with directives on file DIRFILE and
                                           the listing to be written to file FMUOUT.
disfa bindata_nve do=all  <---------------- Requests displaying all file attributes for
                                           file BINDATA_NVE.
delfc $echo echout  <---------------------- Deletes the file connection between $ECHO and
                                           ECHOUT.
delic  <----------------------------------- Deletes the interstate connection invoked by
                                           the CREIC command.
procend bincvt  <-------------------------- Terminates the SCL procedure.
```

Figure 11-4. Procedure BINCVT

A listing of the FMU list file appears in figure 11-5. The list file was written to FMUOUT and obtained by the command:

```
/copy_file  input=fmuout
```

```
LIST FILE OF FMU
SOURCE LIST OF DIRFILE                              NOS/VE FMU V1.0 83081

       0          1 setia bindata mf=c170
       0          2 setoa bindata_nve mf=c180 ed=na
       0          3 creor bindata_nve rpv=no_preset
       0          4       F; F; F; F; F; L; I; I; I
     202          5       creorend

   **** NO DIAGNOSTICS
```

Figure 11-5. FMU List File for Sample Binary File Conversion

### NOS/VE FORTRAN Program That Reads the Migrated File

The NOS/VE FORTRAN job to read the migrated file is shown as an SCL procedure. The sample job is shown in figure 11-6.

```
proc rdexamp <------------------ Begins the SCL procedure.
colt fsource <------------------ Specifies the following program text is to be placed on
                                 file FSOURCE.
      program rdexamp  <-------- Begins the FORTRAN program.
c this program reads the binary
c file we migrated from nos.
      real ray(5)
      integer iray(3)
      logical right
      open (3, file='bindata_nve', form='unformatted')
      rewind 3
      open (7, file='$output')
      do 100 i=1,10
      read (3) ray,right,iray
 100  write (7, fmt='(5f8.3,15,3i5)') ray,right,iray
      close (3)           (fortran program continued.)
      close (7)
      end
** <--------------------------- Terminates collecting text that creates the source
                                 program on file FSOURCE.
attf $user.bindata_nve <-------- Attaches the NOS/VE file.  This command is required to
                                 make the file available to the program if you logged out or
                                 detached the file since it was created.
fortran i=fsource l=ftnout <---- Executes NOS/VE FORTRAN.
lgo  <-------------------------- Executes the FORTRAN binary file.
procend rdexamp <--------------- Terminates the procedure.
```

Figure 11-6. Sample NOS/VE FORTRAN Program RDEXAMP

The listing of the binary file appears the same as the previous listing of file CHRDATA.

## Predefined Collation Tables

NOS/VE has predefined collation tables that you can use in migrating indexed sequential files.

When an indexed sequential file with character keys is created, a collating sequence applies to the file to order the records. You may be concerned with the collating sequence of a file in a migration situation because your application may be dependent on the sequence. This situation can apply either if a special collating sequence is specified for the NOS file or if the NOS default collating sequence applies because the NOS and NOS/VE default sequences are different.

### Considerations

First you should be aware that the NOS and NOS/VE default collating sequences are different. On NOS, the default collating sequence is the CDC character set collating sequence (also, commonly called COBOL6). On NOS/VE, the default collating sequence is ASCII (which is 7-bit ASCII defined by the ANSI X3.4-1977 standard).

When thinking about specifying another collating sequence, consider if the following situations apply to your applications:

- Primary key is alphabetic only (no numbers or other characters). You get the same results no matter which collating sequence you use.
- Primary key can contain digits and letters. You need to evaluate your applications with consideration of the order in which the collating sequence collates these characters. Digits have a lower weight (would collate first) in the ASCII sequence. Letters have a lower weight (would collate first) in the display, COBOL6, and EBCDIC sequences.
- Primary key can contain any character. Each collating sequence assigns different weights to characters. You need to specify the same collating sequence for your migrated file have the same order as the NOS file. In addition, you need to determine how NOS/VE handles characters not available on NOS. (NOS/VE handles this problem with two variants of common collating sequences.)
- A primary key value on NOS was insensitive to uppercase and lowercase. By default on NOS/VE, uppercase and lowercase letters are distinguished. (NOS/VE also handles this potential problem with two varients of common collating sequences.)
- Your applications specify a high key value or low key value to position a file or control processing. You need to consider whether the high or low value in the old sequence has the same effect with the ASCII sequence.
- Your application must process as fast as possible. Processing is fastest using the NOS/VE default (7-bit ASCII) sequence.

For listings of the collating sequences, see appendix C.

### Quick Comparison

The following table compares relative positioning of digits, letters, and some special characters in common collating sequences. When reading the table, assume that each entry in a list is a record key. The different lists show the different ordering records depending on the collating sequences.

| CDC† or COBOL6 | ASCII | Display | EBCDIC |
|---|---|---|---|
| -BLANK | -BLANK | A-ALPHA | -BLANK |
| @-AT | #-POUND | 1-DIGIT | ^-HAT |
| #-POUND | 1-DIGIT | -BLANK | ?-QUEST |
| ?-QUEST | ?-QUEST | #-POUND | #-POUND |
| ^-HAT | @-AT | ?-QUEST | @-AT |
| A-ALPHA | A-ALPHA | @-AT | A-ALPHA |
| 1-DIGIT | ^-HAT | ^-HAT | 1-DIGIT |

†CDC character set collating sequence is the default sequence for indexed sequential files on NOS.

## Concepts for Variants of Common Sequences

There are a number of possible ways to arrange the collation tables to produce different desired effects in the migrated application. Therefore, NOS/VE provides two variants of several common sequences used on NOS. The variants are strict or folded.

### Strict Variant

The strict variant of common collating sequences should be chosen if your application fits the following situation:

> You expect your application to handle only the set of 63 or 64 characters that it knew on NOS. If the application encounters any characters outside this set on NOS/VE, you want the characters to be equivalent to a blank (a space) in any comparison operation.

### Folded Variant

The folded variants of common collating sequences should be chosen if your application fits the following situation:

> Your application is interactive and operates satisfactorily under NOS. The application accepts both uppercase and lowercase letters from the terminal, but because of character set limitations under NOS, these appear in your files as uppercase letters. Now that the limitations of the NOS 63- or 64-character set no longer apply on NOS/VE, you wish your program to continue to operate almost as it did on NOS while enjoying the NOS/VE extended character set. In particular, you want comparison operations that involve old and new data to be case insensitive to uppercase and lowercase alphabetic data; the extra characters are not co-equated (they remain distinct from each other) but collate equal to their counterparts on the NOS set.

## Specifying File Collating Sequence

A collating sequence must be specified in the file information table of the affected file. Two file information fields are involved in designating collating sequence: KEY_TYPE (KT) and COLLATE_TABLE_NAME (CTN). These fields work together as follows for the allowed key types:

KT=I (or KT=INTEGER) Specifies integer keys. No collating sequence or collation table applies.

KT=UC (or KT=UNCOLLATED) The default 7-bit ASCII sequence applies and, therefore, no collation table is designated for the file. You do not specify a value for the COLLATE_TABLE_NAME attribute.

KT=C (or KT=COLLATED) Specifies that a collation table applies. You specify the collation table with the COLLATE_TABLE_NAME attribute.

Standard collating sequences are available for creating files on NOS/VE. All you need to do is specify the particular collation table name as the value of the COLLATE_TABLE_NAME (CTN) attribute. The collation table names follow (the strict and folded NOS/VE variants are listed in association with the corresponding NOS sequence):

DISPLAY63 NOS 6-bit display code sequence when the 63 character set is specified. The tables for NOS/VE are:

```
OSV$DISPLAY63_FOLDED
OSV$DISPLAY63_STRICT
```

DISPLAY64 NOS 6-bit display code sequence when the 64 character set is specified. The tables for NOS/VE are:

```
OSV$DISPLAY64_FOLDED
OSV$DISPLAY64_STRICT
```

COBOL6 The default collating sequence for indexed sequential files on NOS. Also called the CDC character set collating sequence. The tables for NOS/VE are:

```
OSV$COBOL6_FOLDED
OSV$COBOL6_STRICT
```

ASCII6 Also called 6/12 ASCII. The tables for NOS/VE are:

```
OSV$ASCII6_FOLDED
OSV$ASCII6_STRICT
```

EBCDIC6 The 6-bit subset of the EBCDIC character set supported on NOS by COBOL 5 and Sort 5. The tables for NOS/VE are:

```
OSV$EBCDIC6_FOLDED
OSV$EBCDIC6_STRICT
```

NOS/VE provides a predefined collation table to support the full EBCDIC character set. The NOS/VE table name is:

```
OSV$EBCDIC
```

For listings of the collation tables, see appendix C.

Additionally, you can designate any sequence you want. The process is to designate a collation weight for each character in the ASCII character set. See the FORTRAN Language Definition Usage manual for more information about specifying user-defined collation tables. This manual discusses using only the tables listed previously.

## Examples of Specifying a Collation Table

Specifying a file collating sequence occurs at file creation time. You can specify the sequence in a SET_FILE_ATTRIBUTES (SETFA) command. This command would be used for a file being created through the File Management Utility (FMU). A FORTRAN program would usually specify a collating sequence through the FORTRAN file interface.

The examples are as follows:

NOS Indexed Sequential File to Migrate

Creating a 7-Bit ASCII (Default) File with FMU

Creating a COBOL6 Folded File with FMU

Creating a COBOL6 Folded File with FORTRAN

### NOS Indexed Sequential File to Migrate

The following file RABOW (short for rainbow) is an indexed sequential file on NOS (listed as ordered by key). The key consists of 10 characters, where any character in the 6-bit display code set is allowed.

```
1---------10------19   <--- Character position
|---key---|       |
 -BLANK    RED
@-AT       ORANGE
#-POUND    YELLOW
?-QUEST    GREEN
^-HAT      BLUE
A-ALPHA    INDIGO
1-DIGIT    VIOLET
```

The following FILE command describes the file on NOS:

```
FILE,RABOW,FO=IS,ORG=NEW,MRL=20,MNR=20,RT=F,KT=S.
FILE,RABOW,KL=10,EMK=YES,EFC=3,MBL=600,RKP=0,RKW=0.
```

### Creating a 7-Bit ASCII Sequenced File With FMU

The example shows migrating file RABOW (shown in the preceding paragraphs) to NOS/VE as file ASCII_FILE. The example includes all the commands necessary to use FMU to transfer and convert the file after you are processing on NOS/VE. Then the converted file is listed. The description only briefly discusses the commands. For further explanation of using FMU, see the Example of Migrating a Simple NOS Indexed Sequential File to NOS/VE, which appears earlier in this chapter.

```
creic  <------------------------------------ Connects with NOS.
exeic 'ATTACH,RABOW.'
exeic 'FILE,RABOW,FO=IS,ORG=NEW.'
colt dirfile
setia rabow mr=cl70  <---------------------- FMU directive for input file.
setoa ascii_file ed=na  <------------------- FMU directive for output file.
**
setfa f=ascii_file bt=ss fo=is maxrl=20 ..
  rt=f ek=yes kl=10 kp=0 kt=uc ..  <---------Uncollated keys indicate the 7-bit ASCII
                                             sequence applies.
fmu dir=dirfile l=fmuout  <----------------- Executes FMU.
quit  <------------------------------------- Terminates NOS connection.
 -BLANK    RED          <------------------- NOS/VE file ASCII_FILE ordered by the ASCII
#-POUND    YELLOW                           sequence.
1-DIGIT    VIOLET
?-QUEST    GREEN
@-AT       ORANGE
A-ALPHA    INDIGO
^-HAT      BLUE
```

### Creating a COBOL6 Folded File With FMU

The example shows migrating file RABOW to NOS/VE as file NEWBOW. This example shows specifying the COBOL6 folded collation table to be used to set the collation sequence for the file. The attributes for setting collating sequence in the SET_FILE_ATTRIBUTES command are:

```
key_type=collated
collate_table_name=osv$cobol6_folded
```

The example includes all the commands necessary to use FMU to transfer and convert the file after you are processing on NOS/VE. Then the converted file is listed. The description only briefly discusses the commands. For further explanation of using FMU, see the Example Migrating a Simple NOS Indexed Sequential File to NOS/VE, which appears earlier in this chapter.

```
creic  <-----------------------------------Connects with NOS.
exeic 'ATTACH,RABOW.'
exeic 'FILE,RABOW,FO=IS,ORG=NEW.'
colt dirfile
setia rabow mf=cl70  <---------------------- FMU directive for input file.
setoa newbow ed=na  <----------------------- FMU directive for output file.
**
setfa f=newbow bt=ss fo=is maxrl=20  ..
  rt=f ek=yes kl=10 kp=0 kt=c ..  <--------- Specifies collated keys.
  ctn=osv$cobol6_folded  <------------------ Specifies COBOL6 folded table.
fmu dir=dirfile l=fmuout  <---------------- Executes FMU.
quit  <------------------------------------- Terminates NOS connection.
 -BLANK    RED  <-------------------------- NOS/VE FILE NEWBOW ordered by the COBOL6
                                             sequence.
@-AT       ORANGE
#-POUND    YELLOW
?-QUEST    GREEN
^-HAT      BLUE
A-ALPHA    INDIGO
1-DIGIT    VIOLET
```

### Creating a COBOL6 Folded File With FORTRAN

The FORTRAN example includes the FILEIS subroutine call that sets fields in the file information table for file NEWBOW (same file as shown for the previous example). The critical fields for specifying collating sequence are:

| | |
|---|---|
| 'KT', 'S' | Note: KT=C does not work for the FORTRAN file interface. KT=S (for symbolic) designates collated keys. This is for consistency with FORTRAN 5 on NOS. |
| 'CTN', 'OSV$COBOL6_FOLDED' | Designates collation table name. |

The complete CALL FILEIS statement appears as follows:

```
      CALL FILEIS (ISFIT, 'LFN', 'NEWBOW', 'BT', 'SS',
     X 'RT', 'F', 'MRL', 20, 'WSA', REC,
     X 'KT', 'S', 'KL', 10, 'RKP', 0, 'EMK', 'YES',
     X 'KA', REC (1:10), 'KP', 0, 'DFC', 3,
     X 'CTN', 'OSV$COBOL6_FOLDED')
```

# FORTRAN and COBOL File Migration Aids 12

# FORTRAN and COBOL File Migration Aids 12

The FORTRAN and COBOL File Migration Aids enable you to access and migrate your FORTRAN and COBOL data files from NOS to NOS/VE.

## FORTRAN File Migration Aid

The FORTRAN File Migration Aid (FORTRAN FMA) enables a FORTRAN program executing on NOS/VE to access NOS files for two purposes:

- Migrate files, which allows the program to automatically migrate the NOS input files to NOS/VE as the files are read.
- Extended access, which allows the program to read and write NOS files.

The description of using FORTRAN FMA is divided into the following sections:

- Overview of FMA

  Diagrams processing with FMA and discusses general use of FMA.

- File and FORTRAN Requirements

  Details requirements that your files and your program must meet to use FORTRAN FMA.

- Steps in Executing FORTRAN FMA

  Lists the required steps and summarizes them.

- FILE Command for CYBER 170 Files

  Provides detailed information on describing the CYBER 170 files to FMA. (Except for NOS tape files on a NOS/VE tape drive, every CYBER 170 file used with FMA must be described in a FILE command. NOS tape files on a NOS/VE tape drive require a CREATE_170_REQUEST command.)

- FORTRAN FMA Command Descriptions

  Describes command syntax.

- Examples

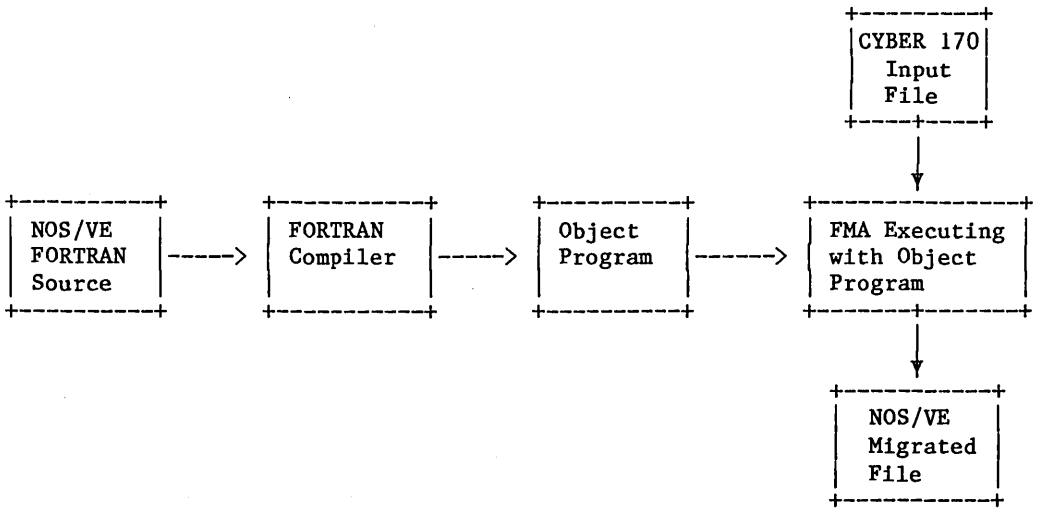
  Shows jobs that create CYBER 170 files and migrates them with FMA.

- Execution Considerations

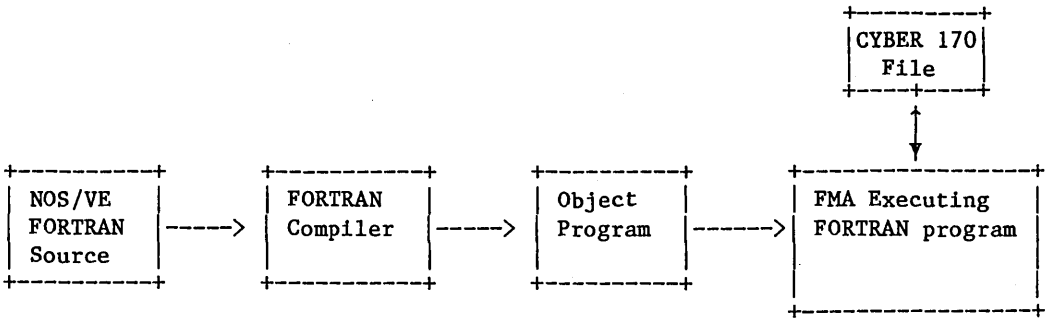
  Provides miscellaneous details about FMA execution and file processing.

## FORTRAN FMA Overview

FORTRAN FMA can execute with regular FORTRAN application programs compiled in the normal way. At execution time, you call FMA and execute the program through FMA. FMA provides for reading or migrating a CYBER 170 file. The following diagram shows the processing flow for migrating a file.



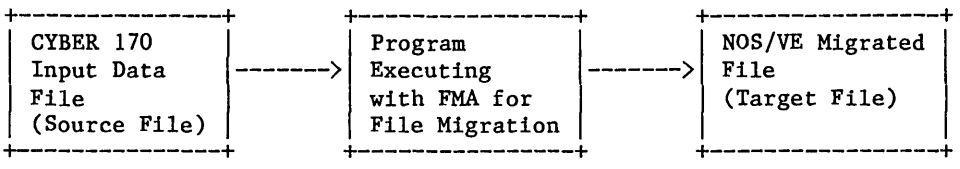

The following diagram shows the processing flow for reading from and writing to a CYBER 170 file:




FMA provides two distinct types of file access: migration and extended access. You specify either type of access for your files when you execute your program through FMA.

### Migration

The following diagram illustrates migration:



Each FORTRAN READ of the source file results in FMA writing a record to the target file. FORTRAN statements do not reference the target file, only the source file. You associate your source and target files in the command that executes your program with FMA.

Migration processing is intended to migrate complete files. When migration begins (when your program opens the file), both the source and target files are positioned at beginning-of-information (BOI). Any data in the target file is lost.

### Extended Access

Extended access allows your FORTRAN program to read or write a NOS file. No migration of the file occurs. Your program can only read the CYBER 170 file.

### Using Regular NOS/VE Files

When using FMA, your program can use any number of NOS/VE files for normal input/output processing. FMA assumes that all files your program references are NOS/VE files unless you explicitly specify that the file is a CYBER 170 file in a command to FMA (namely, EXECUTE_MIGRATION_TASK).

### Using Migration Programs or Subprograms

Although FORTRAN FMA is intended to be used with regular applications programs, there are situations in which you would want to write a FORTRAN program or subprogram to migrate files. This can occur when the applications program does not read a file sequentially or completely or opens and closes the file several times. Because FMA migrates only data that is read, and positions migration files at BOI on each open and close, the regular program cannot migrate the whole file.

When you use a migration subprogram, you can have the subprogram read the entire file (migrating it) before execution of the regular application begins. The read statements in the subprogram must be equivalent with respect to data type and position to the read statements in the applications program. In this situation, input/output statements reference files as follows:

```
Migration subprogram:    READ oldfile
Regular processing:      READ/WRITE migrated-file
```

## File and FORTRAN Requirements for FMA

The FORTRAN program and file being accessed or migrated must meet the following requirements to be used with FMA:

1. The FORTRAN statements used in accessing and migrating a NOS file must provide for one of the following types of input processing:

   - Formatted READ (sequential or direct access)
   - Unformatted READ (sequential or direct access)
   - List directed READ
   - NAMELIST READ

   A brief review of FORTRAN input/output (I/O) follows this list of file and FORTRAN requirements.

   CYBER 170 files used with the following FORTRAN I/O processing cannot be handled by FMA: by buffer I/O, mass storage I/O, and by CYBER Record Manager (CRM) subroutine calls. If your files are of these types, reconsider your migration plan. See Migrating FORTRAN Programs discussion in chapter 14 for suggestions for migrating files of each type.

2. The FORTRAN program must execute on NOS/VE. In other words, the FORTRAN program must be migrated. (For the statements supported by FMA, there are few FORTRAN program migration considerations. However, for information about migrating FORTRAN programs, see chapters 13 and 14.)

3. In both the CYBER 170 and NOS/VE versions of the program, the READ statements used in reading or migrating the CYBER 170 files must be equivalent with respect to data type and position.

4. To migrate the entire file, the general rule with some exceptions is: Only data that the program reads is migrated. The detailed rules for complete file migration are:

   - For formatted, list directed, and NAMELIST sequential files -- each record must be read although all data in each record does not need to be read. For formatted sequential files, data records skipped by means of a slash (/) in the format specification are migrated although not read.

   - For unformatted sequential files -- all data in each record and all records must be read.

   - For formatted direct files -- all records must be read. Reading a partial record causes the whole record to be migrated.

   - For unformatted direct files -- all data in each record and all records must be read.

   These requirements do not apply to a program using CYBER 170 file with the FMA extended access feature.

5. Character data in the file must be in 63- or 64-character display code format.

6. Data of type BOOLEAN can be read and migrated as a bit pattern.

   In general, when boolean data is present, FORTRAN source code changes are probably necessary when migrating the program to NOS/VE. You need to determine your migration strategy. For information about migrating boolean data, see the discussion of boolean data in chapter 14, Migrating FORTRAN Programs. For information about how FMA handles boolean items, see the discussion of Migrating Boolean Items later in this chapter.

These items complete the FORTRAN and file requirements for using FMA.

FMA issues informative messages for conditions it encounters in migrating files. See the FMA CONSIDERATIONS section of this chapter for information on how FMA handles exceptions and file positioning.

## Review of FORTRAN I/O

This review shows code that implements input/output (I/O) supported by FORTRAN FMA. These descriptions should help you identify the kind of I/O used in your program. For more information, see the FORTRAN Language Definition Usage manual.

### Sequential or Direct

In a sequential access file, records are written and read in sequential order. To specify that a file is a sequential access file, the FORTRAN program includes the ACCESS='SEQUENTIAL' specifier on the OPEN statement for the file. If the ACCESS specifier is omitted, the file is assumed to be a sequential access file.

A FORTRAN direct access file provides random access of records (records can be read or written in any order). To specify that a file is a FORTRAN direct access file, the FORTRAN program specifies ACCESS='DIRECT' on the OPEN statement for the file.

### Formatted I/O

FORTRAN formatted I/O is characterized by a format specification used to describe the data being read or written. The following example describes reading 10 real numbers (F descriptor) from unit 5:

```
      READ (5,100) (ARRAY(I),I=1,10)
  100 FORMAT (5F7.2)
```

A formatted I/O applies to both sequential and direct access files.

### Unformatted I/O

FORTRAN unformatted I/O does not use a format specification. The following example reads 10 real numbers from unit 5:

```
REAL ARRAY(10)
    .
    .
READ (5,ERR=99) ARRAY
```

### List Directed I/O

FORTRAN list directed I/O involves the conversion of records according to compiler-defined formatting rules (without an explicit format specification). The following example reads 10 floating point numbers from unit 5:

```
REAL ARRAY(10)
    .
    .
READ (5,*) ARRAY
```

### NAMELIST I/O

NAMELIST I/O permits formatted input of groups of variables and arrays by identifying a group name instead of a format specification. For NAMELIST I/O, a NAMELIST statement must associate a group name used in the READ statement with individual variables or arrays. For example of a NAMELIST READ of unit 5, items QUANT and COST are read in the NAMELIST group VAL:

```
NAMELIST /VAL/QUANT,COST
:
READ (5, VAL)
```

## Steps in Executing FORTRAN FMA

Each of the following steps is required to use FMA.

1. Migrate the NOS FORTRAN 5 program to NOS/VE. (For detailed information about this process, see chapters 13 and 14.)

2. Compile the FORTRAN program on NOS/VE and write the object code to the binary file (LGO is the default file name).

3. For a NOS tape file on a NOS/VE tape drive, enter a CREATE_170_REQUEST command.

4. To ensure correct values for the record type and block type of each output NOS/VE migrated file, use a SET_FILE_ATTRIBUTES command:

```
SET_FILE_ATTRIBUTES  FILE=output file ..
    BLOCK_TYPE=keyword ..
    RECORD_TYPE=keyword
```

For more information about which values to use for BLOCK_TYPE and RECORD_TYPE, see the CYBIL File Interface Usage manual or the FORTRAN Language Definition Usage manual.

5. Call the FMA utility. This call opens the migration environment and enables you to execute the commands available to migrate files. The command is:

```
OPEN_FILE_MIGRATION_AID (OPEFMA)
```

6. Set up a connection to the CYBER 170 to enable the execution of NOS commands. The command is:

```
OPEN_170_STATE (OPE1S)
```

7. Attach and describe the CYBER 170 files by specifying NOS commands in the EXECUTE_COMMAND (EXEC) command as follows:

```
EXECUTE_COMMAND COMMAND='NOS command string.' STATUS=status
```

The NOS command is a string (uppercase or lowercase letters) in the NOS/VE command. Terminate the NOS command with a period. Enclose the NOS command in apostrophes. The STATUS parameter is optional in the command. (NOTE: Never execute a NOS RETURN,* or CLEAR command while FMA is active; doing so returns files used by FMA causing FMA to fail. Executing an EXIT control statement also causes FMA to fail.)

7.1. For a CYBER 170 disk file, make the file local by executing NOS commands such as GET or ATTACH. For example, attaching file AFILE:

```
FA/execute_command  command='ATTACH,AFILE/PW=XYZ.'
```

For a tape file to be migrated through the interstate connection, execute a NOS REQUEST command. For example:

```
FA/execute_command  command='REQUEST,...'
```

7.2 Describe your file with a FILE control statement, or, for a CYBER 170 tape file on a NOS/VE tape drive, with a CREATE_170_REQUEST command. For example, disk file AFILE is used with formatted, sequential FORTRAN input/output:

```
FA/execute_command  command='FILE,AFILE,FO=SQ,RT=Z,BT=C,MRL=170.'
```

7.3 Repeat steps 7.1 and 7.2 for as many CYBER 170 files as your application uses.

8. Close the connection to the CYBER 170. You must close this connection before you can execute the program. The command is:

```
CLOSE_ENVIRONMENT (CLOE or QUIT)
```

9. Specify executing the migration task. The command (simplified) is:

```
EXECUTE_MIGRATION_TASK ..
   MIGRATION_FILES=((nosfile-1,nvefile)) ..  <---- To migrate
   EXTENDED_ACCESS_FILES=((nosfile-2,C170)) .. <-- To read/write only
   FILE=LGO ..  <------------------------------- Executes program
   PARAMETER='string'  <------------------------ Passes parameters
```

The PARAMETER parameter specifies parameters to be passed to the program (the parameters that normally appear with LGO).

Specify the EXECUTE_MIGRATION_TASK command for as many migration programs as you wish to execute.

10. Terminate use of FORTRAN FMA. The command is:

```
CLOSE_ENVIRONMENT (CLOE or QUIT)
```

**Summary of the Steps Required in Using FMA**

1. Migrate the NOS FORTRAN 5 program to NOS/VE.
2. Compile the FORTRAN program on NOS/VE.
3. For a CYBER 170 tape file on a NOS/VE disk drive, execute a CREATE_170_REQUEST command.
4. Enter a SET_FILE_ATTRIBUTES command to ensure the correct values for the NOS/VE migrated file for block type and record type.
5. Enter command: OPEN_FILE_MIGRATION_AID (OPEFMA)
6. Enter command: OPEN_170_STATE (OPE1S)
7. For CYBER 170 disk files, make the files local and describe them in a FILE command. For example:

```
FA/execute_command  command='ATTACH,AFILE/PW=XYZ.'
FA/execute_command  command='FILE,AFILE,FO=SQ,RT=Z,BT=C,MRL=170.'
```

For CYBER 170 tape files to be migrated through the interstate connection, execute a REQUEST command and describe the files in a FILE command.

Repeat the commands for as many CYBER 170 files as your program reads.

8. CLOSE_ENVIRONMENT (CLOE or QUIT)
9. Execute the migration task. The command is:

```
EXECUTE_MIGRATION_TASK ..
   MIGRATION_FILES=((nosfile-1,nvefile)) ..  <----- To migrate
   EXTENDED_ACCESS_FILES=((nosfile-2,C170)) ..  <-- To read/write only
   FILE=LGO ..  <------------------------------- Executes program
   PARAMETER='string'  <------------------------ Passes parameters
```

10. CLOSE_ENVIRONMENT (CLOE or QUIT)

## File Command for CYBER 170 Files

The CYBER 170 files must be described in a FILE command. A FILE command provides information for a file information table (FIT) used by CYBER Record Manager to process the file.

If the job stream for your FORTRAN program includes FILE commands (also called FILE control statements), you can use information from those FILE commands in your FMA job. If, however, your FORTRAN program uses default file information, you need to determine the appropriate file information according to the type of input/output processing for the file.

Information for FILE commands is described in this chapter in topics as follows:

Required FILE command information

Typical FILE commands

Determining record length

Background information on CYBER 170 file is available in appendix E, FORTRAN Default FIT Fields. This appendix lists default values provided for files by FORTRAN on NOS.

### Required File Command Information

The following fields must be specified on the FILE command unless the default value is appropriate (default values supplied by NOS CYBER Record Manager apply). (For NOS tape files on a NOS/VE tape drive, file attributes are specified on the CREATE_170_REQUEST command.)

FILE,lfn,FO=org,RT=type,BT=type,MRL=length,EO=A.

lfn File name (on the CYBER 170).

FO File organization. The organizations are SQ for sequential (default) or WA for word addressable.

RT Record type. The default value is W (control word).

BT Block type. The default value is I (internal blocking).

MRL Maximum record length in characters. Only direct access files require the exact record length; for others, some length equal to or longer than the longest record works. FL can be specified instead of MRL.

EO Error options. EO=A for accept bad data is recommended for tape files so that the data is accepted even though parity errors occur. The FORTRAN default is similar; however, the CYBER Record Manager default used by FMA is T (terminate). The CRM default works for disk files.

The following file information must apply to the CYBER 170 file, but usually the default values are appropriate:

IC Internal code. 63-character display code (IC=D63) or 64-character display code (IC=D64).

These fields are not documented in the CYBER Record Manager Basic Access Methods reference manual but have been added to the file information table for file migration. You should not be concerned about the values for these fields unless file processing at your site changes the default values.

### Typical File Commands

If your NOS FORTRAN program did not specify a file command for the files you are migrating, the file commands required to describe the files to FMA are shown below.

| FORTRAN I/O | FILE command |
|---|---|
| Sequential I/O:<br>Formatted<br>NAMELIST<br>List Directed | FILE,lfn,FO=SQ,RT=Z,BT=C,EO=A,MRL=length. |
| Sequential I/O:<br>Unformatted | FILE,lfn,FO=SQ,RT=W,BT=I,EO=A,MRL=length.<br>or using CRM defaults: FILE,lfn,EO=A,MRL=length. |
| Direct Access I/O:<br>Formatted and<br>Unformatted | FILE,lfn,FO=WA,RT=U,BT=C,EO=A,MRL=length. |

lfn - CYBER 170 file name.

length - Maximum record length (CYBER 170 length in characters). Exact length for direct access files. Exact length or greater for sequential files.

You need to determine the appropriate record length for the files you are migrating. The information on determining record length follows.

### Determining Record Length

To determine record length, you need to determine the space the data in an input/output list requires. Assume your program describes and reads data as follows:

```
CHARACTER*9 C(4)
REAL RAY(5)
INTEGER IRAY(3)
LOGICAL RIGHT
OPEN (2, ACCESS='DIRECT', FORM='UNFORMATTED',RECL=??)
READ (2) C, RAY, IRAY, RIGHT
```

Data read when the READ statement executes makes up a record (the data items make up the list).

When using FMA to read or migrate a file, you must determine record length in CYBER 170 figures to specify in the FILE command. Additionally, if you are using direct access files, you need to specify record length in the RECL parameter (FORTRAN OPEN statement) in lengths for NOS/VE characters or words.

The record length for NOS must include the space required for word alignment. Calculating record length on NOS is easiest if you calculate the number of words.

On NOS/VE, word alignment is not a consideration. Calculating record length is easiest if you calculate the number of characters.

CYBER 170 words contain ten 6-bit bytes (10 characters). NOS/VE words contain eight 8-bit bytes (8 characters).

For example, record length is figured for a NOS/VE FORTRAN program that uses data items as follows:

| | CYBER 170 Words | NOS/VE Characters |
|---|---|---|
| CHARACTER*9 C(4) | 4† | 36 |
| REAL RAY(5) | 5 | 40 |
| INTEGER IRAY(3) | 3 | 24 |
| LOGICAL RIGHT | 1 | 8 |
| OPEN (2,ACCESS='DIRECT',FORM='UNFORMATTED',RECL=14) | | |
| READ (2,REC=1) C, RAY, IRAY, RIGHT | | |

†C word length (9*4 = 36 + 9 = 45. Divide by 10, drop remainder = 4 words)

The record length is:

CYBER 170: 13 words (13 times 10 characters/word = 130 characters)

NOS/VE: 108 characters (108+7=115. Divide by 8, drop remainder= 14 words)

For the NOS FILE command, the record length is specified in characters:

```
/execute_command  command='FILE,filename,FO=WA,RT=U,BT=C,EO=A,MRL=130.'
```

### Record Length Rules for CYBER 170

1. Count one word for each noncharacter item except for double precision and complex items, which count as two words.
2. Calculate the length in words of each continuous group of characters by adding 9 to the combined length of the items in characters, dividing the result by 10, truncating the fractional part.
3. For the record length in words, add the sums resulting from steps 1 and 2.
4. For record length in characters, multiply the sum from step 3 by 10.

When using FMA, you need record length in characters for MRL (or FL) specified in the FILE command.

### Record Length Rules for NOS/VE

1. Count each noncharacter item as 8 characters except for complex and double precision items, which count as 16 characters.
2 Calculate the total number of characters in all the character items.
3. For the record length in characters, add the lengths calculated in rules 1 and 2.
4. For the record length in words, use the sum calculated in rule 3, add 7, divide the result by 8, and truncate the fractional part. (The record length in words is required only for the FORTRAN OPEN statement for unformatted direct I/O.)

## FORTRAN FMA Command Descriptions

This section provides and overview and detailed descriptions of FORTRAN FMA commands.

The syntax of FORTRAN FMA commands follows the System Command Language (SCL) conventions. (For an introduction to SCL command conventions, see chapter 2.)

The command formats use an SCL convention for showing syntax. Each appears on a separate line below the command. If the parameters were listed in this way in use, an ellipsis must appear at the end of each continued line.

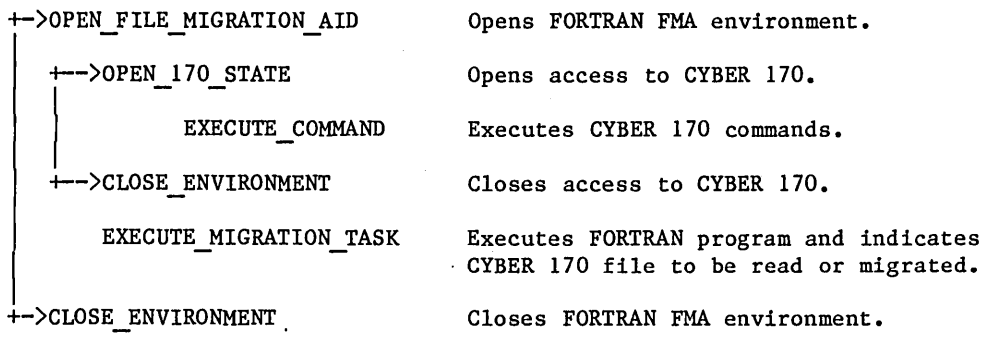
The commands are:

```
OPEN_FILE_MIGRATION_AID
OPEN_170_STATE
CLOSE_ENVIRONMENT
EXECUTE_COMMAND
EXECUTE_MIGRATION_TASK
```

FORTRAN FMA is a NOS/VE command utility. The OPEN_FILE_MIGRATION_AID command activates the utility. When using the utility, you can issue any NOS/VE command.

### Overview of FORTRAN FMA Commands

FORTRAN FMA commands must be entered in a particular order because the commands provide a processing path through a migration environment. Each command either opens a specialized processing environment, executes an operation in the environment, or closes an environment. The following diagram indicates execution order; commands can be executed only within the containing bracket.




### OPEN_FILE_MIGRATION_AID (OPEFMA)

The OPEN_FILE_MIGRATION_AID (OPEFMA) command opens the file migration environment. This command must be executed to use any other FORTRAN FMA command.

Format:
```
OPEN_FILE_MIGRATION_AID
    STATUS=variable
```

Parameter: Status variable; optional parameter. See the discussion of the status variable in chapter 4.

After this command is executed, the system prompts for input as follows: FA/

```
Example:    /open_file_migration_aid
            FA/
      or
            /opefma
            FA/
```

### OPEN_170_STATE (OPE1S)

The OPEN_170_STATE command opens access to the CYBER 170 and enables the user to execute NOS commands.

```
Format:     OPEN_170_STATE
                STATUS=variable
```

Parameter: Status variable; optional parameter. See the discussion of the status variable in chapter 4.

```
Example:    FA/open_170_state
      or
            FA/opeis
```

### CLOSE_ENVIRONMENT (CLOE)

The CLOSE_ENVIRONMENT (CLOE or QUIT) command closes the current environment so that subsequent commands are executed in the containing environment.

```
Format:     CLOSE_ENVIRONMENT
                STATUS=variable
```

Parameter: Status variable; optional parameter. See the discussion of the status variable in chapter 4.

```
Example:    /open_file_migration_aid
            FA/open_170_state
             :
            FA/close_environment  <----- Closes the CYBER 170 connection but remains in the
                                         FORTRAN FMA utility.
```

### EXECUTE_COMMAND (EXEC)

The EXECUTE_COMMAND command provides for executing a NOS command.

You can specify one or more NOS commands in EXECUTE_COMMAND. Any diagnostics issued by the NOS commands are written to the NOS/VE job log. Thus, if an error occurs, you can determine which NOS command caused the error by examining the job log. Use the DISPLAY_LOG command to display the job log.

CYBER loader sequences are processed as a single command; therefore, the entire sequence must be entered as a list of NOS commands in one EXECUTE_COMMAND.

```
Format:     EXECUTE_COMMAND
                COMMAND='nos-command.'
                STATUS=variable
```

### EXECUTE_COMMAND (EXEC) Parameters

Parameters:

COMMAND — A NOS command entered as a string, terminated with a period, and enclosed in apostrophes. Maximum string length is 80 characters. For example, attaching file AFILE:

```
FA/execute_command  command='ATTACH,AFILE/PW=CAT.'
```

Optionally, NOS commands can appear as a list of strings. NOS receives each string as a separate line. For example:

```
FA/execute_command  command=('LDSET=OLIB.' ..
                             'LGO.')
```

If this command is used in an SCL procedure, SCL can perform parameter substitution. The command is passed to NOS after substitutions occur.

Because FMA uses some NOS local files as working files, you must not issue NOS commands that might affect these files. For example, do not execute a NOS CLEAR or RETURN,* command or an EXIT control statement.

STATUS — Status variable; optional parameter. See the discussion of the status variable in chapter 4.

### EXECUTE_COMMAND (EXEC) Examples

Specifying commands with the parameter name:

```
FA/execute_command  command='ATTACH,DATAFIL/UN=OTHNAME.'
FA/exec c='FILE,DATAFIL,FO=SQ,RT=Z,BT=C,EO=A,FL=2000.'
```

Specifying commands by position:

```
FA/execute_command  command='ATTACH,FILE1/UN=OTHNAME.'
FA/exec 'ATTACH,FILE2.'
```

## EXECUTE_MIGRATION_TASK (EXEMT)

The EXECUTE_MIGRATION_TASK (EXEMT) command specifies the files to be read/written, or to be migrated, and executes the FORTRAN object program.

This command is much like the NOS/VE EXECUTE_TASK command: most of the EXECUTE_TASK parameters are available on EXECUTE_MIGRATION_TASK; the EXECUTE_MIGRATION_TASK includes two additional parameters, which are the MIGRATION_FILES and EXTENDED_ACCESS_FILES parameters.

```
Format:       EXECUTE_MIGRATION_TASK
                  MIGRATION_FILES=((c170-file-name,c180-file-name,C170_TO_C180))
                  EXTENDED_ACCESS_FILES=((file-name,keyword))
                  FILE=list-of-files
                  PARAMETER='string'
                  LIBRARY=list-of-files
                  MODULE=list-of-modules
                  STARTING_PROCEDURE=program-name
                  LOAD_MAP=file
                  LOAD_MAP_OPTION=list-of-keywords
                  PRESET_VALUE=keyword
                  TERMINATION_ERROR_LEVEL=keyword
                  STACK_SIZE=integer
                  DEBUG_INPUT=file
                  DEBUG_OUTPUT=file
                  ABORT_FILE=file
                  DEBUG_MODE=boolean
                  STATUS=status variable
```

### EXECUTE_MIGRATION_TASK (EXEMT) Parameters

Two parameters are unique to EXECUTE_MIGRATION_TASK and provide capabilities as follows:

**MIGRATION_FILES** Specifies the names of one or more files to be migrated as they are read by your program. This parameter also specifies the direction of migration, which must be CYBER_170_to_CYBER_180. Each item of the list is of the form:

```
((c170-file-name,c180-file-name,C170_TO_C180))
or
((c170-file-name,c180-file-name))
```

where c170-file-name is the name of a CYBER 170 file and c180-file-name is the name of a CYBER 180 file. C170_TO_C180 is a keyword specifying the direction of migration; no other keywords are available and C170_TO_C180 is the default.

For CYBER 170 tape files on a NOS/VE tape drive, c170-file refers to a temporary NOS/VE file associated with a NOS tape file by a previous CREATE_170_REQUEST command. c170-file and c180-file must not be the same name; if they are, FMA issues an error message and terminates.

**EXTENDED_ACCESS_FILES** Specifies a file that is to be read or written by your program, but is not to be migrated. The value can be a list; each item can be in one of these forms:

```
c170-file-name
(c170-file-name,C170)
(c180-file-reference,C180)
```

where c170-file-name and c180-file-name specify the CYBER 170 and CYBER 180 files to be read or written. C170 and C180 are keywords that indicate if the file you specified is a CYBER 170 or 180 file.

For CYBER 170 tape files on a NOS/VE tape drive, c170-file refers to a temporary NOS/VE file associated with a NOS tape file by a previous CREATE_170_REQUEST command. c170-file and c180-file must not be the same name; if they are, FMA issues an error message and terminates.

The following table indicates the possible combinations that can be specified for the EXTENDED_ACCESS_FILES (EAF) parameter; for each combination, the table shows whether a 170 or 180 file is used.

| | C170 keyword specified | C180 keyword specified | No keyword specified |
|---|---|---|---|
| File name specified | EAF file is a 170 file | EAF file is a 180 file | EAF file is a 170 file |
| File name not specified | error | error | error |

If EXTENDED_ACCESS_FILES is not specified for a file, the file is a 180 file.

The other parameters are common to the EXECUTE_TASK command and work the same way in both commands. They are:

FILES (F) Specifies either the object files or object library files whose modules are unconditionally loaded. A module is object code acceptable to the loader: in this situation, a compiled FORTRAN program or subprogram. This parameter is optional.

If FILE is omitted, the STARTING_PROCEDURE parameter specifies the object module at which execution begins.

If both FILE and STARTING_PROCEDURE parameters are omitted, the loader attempts to execute an object program on file $LOCAL.LGO.

PARAMETER (P) Optional parameter that provides a method of passing values to the executing program. The parameter is just a string to be added to the execution command. Enclose the complete parameter string in apostrophes '..'.

If a list of parameters is specified, the parameters must conform to the format for parameter lists described in the FORTRAN Language Definition Usage manual.

Three classes of parameters can appear in this parameter:

a. File Name Substitution Parameters

b. System Command Language Parameters

c. $PRINT_LIMIT

The $PRINT_LIMIT and either SCL parameters or file name substitution parameters (but not both) can appear on a given execution command.

In the PARAMETERS parameter, you can substitute new file names for existing file names at execution time just as you can specify file names on a name call execution of your FORTRAN program. (The same rules apply in both situations for file name substitution.) File names specified as parameters are substituted for file names associated with unit names declared on the PROGRAM statement in the FORTRAN program.

If you refer to the same file in the PARAMETERS parameter and in other parameters of the EXECUTE_MIGRATION_TASK command, specify the same file name in all parameters.

The target file in the MIGRATION_FILES command is not affected by file name substitution. The target file is known to the system only by the file reference specified in the MIGRATION_FILES parameter.

Unit names declared in the PROGRAM statement are associated with a default file name unless you substitute a different name. For units INPUT and OUTPUT, the default files are $INPUT and $OUTPUT, respectively. For other units, the default files have the same name as the unit. For example:

```
PROGRAM TEST (INPUT, OUTPUT, TAPE1, TAPE2)
```

Each unit declaration in the PROGRAM statement defines a valid parameter that can appear in the PARAMETER parameter. These parameters are specified on the execution command in the form:

```
PARAMETER='unitname=newname'
```

For example: `parameter='tape1=filea'`

The file name specified by newname (FILEA) is substituted for the file name associated with the specified unit name.

In the PARAMETERS parameter, you can specify parameters that provide a method of passing information between an executing program and the System Command Language (SCL). The parameter names and values are accessed within the program through the SCL subprogram calls.

Any SCL parameters appearing in the PARAMETERS parameter must have been defined by a C$ PARAM directive in the source program. This directive establishes a detailed definition of the parameters. A parameter specified on an execution command must conform to its definition as declared in the C$ PARAM directive.

SCL parameters have the general form:

```
PARAMETERS='parameter name=(list-of-values)'
```

SCL parameters cannot be used with file substitution parameters.

Refer to the SCL Language Definition Usage manual for a detailed description of SCL parameters. Refer also to the FORTRAN for NOS/VE Language Definition Usage Manual for descriptions of the SCL subprogram calls and examples of SCL parameters.

The $PRINT_LIMIT parameter specifies the maximum number of print lines that the executing program can write to files $OUTPUT and $ERRORS. The $PRINT_LIMIT parameter appears in the execution command as follows:

```
PARAMETER='$PRINT_LIMIT=lim'  or
P='$PL=lim'
```

The parameter $PRINT_LIMIT and its value are entered as a string; lim is the desired print limit (decimal number of lines).

For example, the following command begins execution of the a program on LGO and sets the runtime print limit to 10000 lines.

```
FA/execute_migration_task  extended_access_files=nosfile ..
    file=lgo ..
    parameter='$print_limit=10000'
```

| | |
|---|---|
| LIBRARY (L) | Specifies the libraries to be added to the program library list for this execution. These libraries remain in effect only for the current EXECUTE_MIGRATION_TASK command. These libraries remain in effect only for the current EXECUTE_MIGRATION_TASK command and are searched before any libraries in the job library list. |
| MODULE (M) | Specifies a list of modules to be unconditionally loaded from object libraries in the program library list. The modules are loaded in the order they are specified in the list. |

STARTING_PROCEDURE (SP)

Specifies the name of the entry point where execution begins. This is usually the program name. Optional parameter.

If this parameter is omitted, the first module residing in the file specified by the FILE parameter is used. ($LOCAL.LGO is the default file.)

This parameter works in the same way as the STARTING_PROCEDURE parameter of the EXECUTE_TASK_COMMAND. See the SCL Object Code Management Usage manual for more information.

LOAD_MAP

Specifies the file to which the load map is to be written; this also specifies the positioning of the load map file.

LOAD_MAP_OPTION (LMO)

Specifies what information is to be included in the load map. The values you can specify for LMO are:

ALL -- all of the following information is to be included

NONE -- no load map is to be written

SEGMENT -- segment map

ENTRY_POINT -- entry point map

CROSS-REFERENCE -- entry point cross-reference map

PRESET_VALUE (PV)

Specifies the value to be stored in all unused data words. The values you can specify for PV are:

ZERO -- all zeros

FLOATING_POINT_INDEFINITE -- floating-point indefinite value

INFINITY -- floating-point infinite value

TERMINATION_ERROR_LEVEL (TEL)

Specifies the level of error that, if encountered during loading, will terminate program execution. The values you can specify for TERMINATION_ERROR_LEVEL are:

WARNING -- terminates when a warning, error, or fatal error occurs

ERROR -- terminates when an error or fatal error occurs

FATAL -- terminates when a fatal error occurs

STACK_SIZE (SS)

Specifies the upper limit in bytes of the run-time stack used for procedure call linkages and local variables. The value you specify is rounded up to the nearest real page size boundary.

DEBUG_INPUT (DI)

Specifies the file from which Debug is to read its subcommands. These subcommands are executed only when DEBUG_MODE=ON.

DEBUG_OUTPUT (DO)

Specifies the file to which Debug is to write its output. Output is written only when DEBUG_MODE=ON.

ABORT_FILE (AF)

Specifies the file containing the debug commands to be executed if the program aborts. These commands are executed only if the program is not executed in debug mode.

| | |
|---|---|
| DEBUG_MODE (DM) | Indicates if the program is to be executed in debug mode. The values you can specify for DEBUG_MODE are:<br><br>ON -- program is executed under debug control<br><br>OFF -- program is executed without debug control |
| STATUS | Status variable; optional parameter. |

### EXECUTE_MIGRATION_TASK (EXEMT) Examples

FORTRAN program on source file BIGJOB (object file LGO) reads CYBER 170 file BINDATA and migrates it to NOS/VE file NEWFILE:

```
/fortran input=bigjob
/set_file_attributes file=$user.newfile record_type=variable block_type=system_specified
/open_file_migration_aid
FA/open_170_state
FA/execute_command command='ATTACH,BINDATA.'
FA/execute_command command='FILE,BINDATA,FO=SQ,RT=W,BT=I,MRL=90.'
FA/close_environment
FA/execute_migration_task, ..
    migration_files=((bindata,$user.newfile,c170_to_c180)) ..
    file=lgo
FA/close_environment
```

FORTRAN program reads CYBER 170 files FLIGHT and AIRPORT which are specified in the EXTENDED_ACCESS_FILES parameter. The object program resides on file BINARY. The command is:

```
FA/execute_migration_task  extended_access_files=((flight,c170),(airport,c170)) file=binary
```

FORTRAN program (as above) reads and migrates the files specified in the MIGRATION_FILES parameter:

```
FA/execute_migration_task  migration_files=((flight,$user.flight),(airport,$user.airport)) ..
        file=binary
```

The following command begins execution of a program in object file LGO (specified by parameter position), sets the runtime print limit to 10000 lines, and accesses CYBER 170 file NOSFILE as an extended access input file (no migration):

```
FA/execute_migration_task  extended_access_file=nosfile  lgo ..
    parameter='$print_limit=10000'
```

File Name Substitution Example. Program SIMULAT on object file LGO declares units TAPE1 and TAPE2. File names A and F are substituted for the NOS files associated with units TAPE1 and TAPE2. NOS/VE file names are not affected by the substitution; they retain the names designated in the MIGRATION_FILES parameter ($USER.AIRPORT and $USER.FLIGHT).

```
      PROGRAM SIMULAT (TAPE1,TAPE2)
      :
/open_file_migration_aid
FA/open_170_state
FA/execute_command command='GET,A.'  <---------------- Specifies file name known on CYBER 170.
FA/execute_command command='GET,F.'
FA/execute_command command='FILE,A,FO=WA,RT=U,BT=C,EO=A,MRL=110.'
FA/execute_command command='FILE,F,FO=SQ,RT=Z,BT=C,EO=A,FL=20.'
FA/close_environment
FA/execute_migration_task  mf=((a,$user.airport),(f,$user.flight)) ..
        file=lgo, parameter='tape1=a,tape2=f'
FA/cloe
```

## FMA Examples

FORTRAN FMA migration examples are:

Migrating an unformatted sequential file

Migrating two files (sequential list directed file and unformatted direct file)

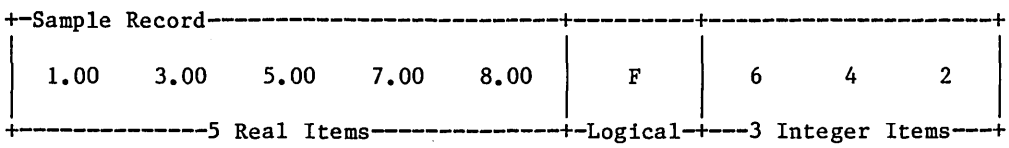
### Migrating an Unformatted Sequential File

This example illustrates migrating a CYBER 170 binary file BINDATA. The example consists of a FORTRAN 5 program that creates the file and the NOS/VE migration task that migrates the file.

File BINDATA contains items declared and written as follows:

```
                                  CYBER 170 Length

     REAL RNUMS(5)                5 words
     LOGICAL TRUTH                1 word
     INTEGER INUMS(3)             3 words
     :                            --------
     WRITE (2) RNUMS,TRUTH,INUMS  9 words (90 characters)
```




The following NOS procedure creates the permanent unformatted sequential file BINDATA.

```
.PROC,RUNKENX.
DEFINE,BINDATA.  <------------------------------ Defines NOS direct access file.
FTN5,I=SOURCE,L=FTN5OUT,REW.
LGO.
.DATA,SOURCE
      PROGRAM WREXAMP
      REAL RNUMS(5)
      LOGICAL TRUTH
      INTEGER INUMS(3)
      DATA RNUMS /1.0,3.0,5.0,7.0,8.0/
      DATA TRUTH /.FALSE./
      DATA INUMS /6,4,2/
      OPEN (2, FILE='BINDATA', FORM='UNFORMATTED')
      DO 10 I=1,10
  10  WRITE (2) RNUMS,TRUTH,INUMS
      CLOSE (2)
      END
```

The task to migrate file BINDATA to NOS/VE file NEWFILE includes commands and the FORTRAN program. The task is shown in figure 12-1.

```
create_file_connection $echo output  <---------------- Displays executed commands.
collect_text source  <------------------------------- Creates FORTRAN source file.
      program migl
C This program reads the binary file
C created on NOS.  Run with FMA, the
C program migrates the file.
      real ray(5)
      integer iray(3)
      logical right
      open (3, file='bindata', form='unformatted')
      open (7, file='$output')
      rewind 3
      do 100 i=1,10
      read (3) ray,right,iray  <---------------------- Causes file migration.
 100  write (7, fmt='(5f8.3,15,3i5)') ray,right,iray
      close (3)
      close (7)
      end
**  <------------------------------------------------- Terminates source file text.
fortran i=source e=ftnout l=ftnout  <----------------- Compiles the FORTRAN program.
"  Start FMA  "
setfa $user.newfile bt=ss rt=v
open_file_migration_aid  <---------------------------- Calls FORTRAN FMA.
open_170_state
execute_command 'ATTACH,BINDATA.'  <------------------ Attaches NOS file BINDATA.
exec 'FILE,BINDATA,FO=SQ,RT=W,BT=I,MRL=90.'  <-------- Describes the file for FMA.
close_environment
execute_migration_task ..   <------------------------- Specifies execution with FMA
  mf=((bindata,$user.newfile,c170_to_c180)) ..  <----- Associates files
  file=lgo  <----------------------------------------- Executes file LGO.
cloe  <--------------------------------------------- Closes FMA
delete_file_connection $echo output
```

Figure 12-1. Migration Task for Unformatted, Binary File.

Migration occurs as the program reads BINDATA. To show the NOS file being read, the program writes the values read from BINDATA to file OUTPUT. (To write to OUTPUT, the program uses the automatic connection between $OUTPUT and OUTPUT. For information about standard files, see chapter 4.)

Commands in the task include several processing steps. CREATE_FILE_CONNECTION specifies a file connection between standard file $ECHO and file OUTPUT so that commands are displayed at the terminal as the task executes. COLLECT_TEXT writes the FORTRAN source code to file SOURCE. Other commands specify FORTRAN compilation and using FORTRAN FMA.

You can run the task by putting the commands and program as listed in a NOS/VE file. Specify execution of the file by entering the INCLUDE_FILE command. For example, to execute a file named FMAJOB1, specify:

```
/include_file  file=fmajob1
```

### Migrating Two Files (Sequential List Directed and Unformatted Direct Access)

This example assumes that an application on NOS computes statistics about test flights. The application requires two files:

FORTRAN direct access file containing the location of airports around the word.

Sequential file containing departure and arrival airports for test flights.

The program obtains the departure and arrival airports from file FLIGHT and accesses location information about the airports from file AIRPORT. The program is then assumed to use the data for statistical analysis (not shown).

This comprehensive example is broken down into the following parts:

Description of the FORTRAN direct access file (AIRPORT)

FORTRAN direct access file creation job

Description of the list directed file (FLIGHT)

Two-file migration task using a subprogram

#### Description of the FORTRAN Direct Access File

A record in file AIRPORT contains the following information: airport code, terminal name, city, state, latitude, and longitude. For example:

```
SFO SAN FRANCISCO INTL AIRPORT    SAN FRANCISCO       CALIFORNIA, U.S.A.
    037.75 N 122.45 W
```

In AIRPORT, however, latitude and longitude are contained in two arrays:

LL where LL(1) is latitude and LL(2) is longitude
D where D(1) is latitude direction and D(2) is longitude direction.

The data in file AIRPORT is declared and written as follows:

```
                                                CYBER 170 Length

      CHARACTER CODE*3,                          3 characters
      CHARACTER TERMNL*29                       29 characters
      CHARACTER CITY*19                         19 characters
      CHARACTER STATE*25                        25 characters
                                       Total    76 characters -->  8 words
      REAL LL(2)                                                  2 words
      CHARACTER*1 D(2)                           2 characters -->  1 word
      :                                         ------------------------
10    WRITE(2,REC=I)CODE,TERMNL,CITY,STATE,LL,D                   11 words
```

The FILE command required to describe this file for execution with FMA is:

```
FILE,AIRPORT,FO=WA,RT=U,BT=C,MRL=110.
```

The maximum record length (MRL) must be specified in characters. A record in AIRPORT requires 11 words. (A word equals 10 characters.)

### FORTRAN Direct Access File Creation Job

The job to create the direct access file includes the program source file, the input data file, and the commands. Assuming the program is in file SOURCE and the data is in indirect access file TAPE1, the following commands compile and execute the program:

```
FORTRAN I=SOURCE.
GET,TAPE1.
LGO.
```

#### Program Listing (on File SOURCE)

```
      PROGRAM CREWAF  (TAPE1,TAPE2)
C  CREATES A WORD ADDRESSABLE FILE FOR DIRECT ACCESS I/O.
C  ONLY EVEN NUMBER RECORDS ARE WRITTEN TO SHOW RANDOM ACCESS.
C
      CHARACTER CODE*3, TERMNL*29
      CHARACTER CITY*19, STATE*25
      REAL LL(2)
      CHARACTER*1 D(2)
      OPEN(2,STATUS='NEW',FILE='AIRPORT',ACCESS='DIRECT',RECL=11)
      DO 10 I=2,14,2
      READ (1,40)CODE,TERMNL,CITY,STATE,LL(1),D(1),LL(2),D(2)
 40   FORMAT (A3,1X,A29,1X,A19,1X,A25/3X,2(1X,F6.2,1X,A1))
 10   WRITE(2,REC=I)CODE,TERMNL,CITY,STATE,LL,D
      CLOSE (2)
      PRINT *, ' CREWAF COMPLETE'
      END
```

#### Listing of Data File TAPE1 (Input for AIRPORT)

```
FRA FRANKFURT-MAIN INTL AIRPORT   FRANKFURT AM MAIN   WEST GERMANY
    050.10 N 008.68 E
LHR HEATHROW AIRPORT-LONDON       LONDON              ENGLAND
    051.47 N 000.45 W
MEL MELBOURNE AIRPORT             MELBOURNE           VICTORIA, AUSTRALIA
    037.75 S 144.97 E
ORY PARIS ORLY AIRPORT            PARIS               FRANCE
    048.87 N 002.03 E
FCO FIUMICINO AEROPORTO-ROME      ROME                ITALY
    041.83 N 012.33 E
SFO SAN FRANCISCO INTL AIRPORT    SAN FRANCISCO       CALIFORNIA, U.S.A.
    037.75 N 122.45 W
HND TOKYO INTL AIRPORT/HANEDA     TOKYO               JAPAN
    035.67 N 139.75 E
```

### Description of the List Directed File

The records in list directed file FLIGHT consist of airport pairs representing the departure and arrival airport for a test flight. Each airport is represented by the airport code and the record number for the particular airport in file AIRPORT. The data in FLIGHT is as follows:

| Arrival-Departure | CYBER 170 Length |
|---|---|
| MEL 06 HND 14 | 13 characters --> 2 words |
| HND 14 SFO 12 | |
| SFO 06 LHR 04 | |
| LHR 04 FRA 02 | |
| ORY 08 FCO 10 | |

The FILE command required to describe this file for execution with FMA is:

```
FILE,FLIGHT,FO=SQ,RT=Z,BT=C,FL=20.
```

The field length (FL or MRL) is specified in characters.

### Two-File Migration Task Using a Subprogram

The task to migrate CYBER 170 files AIRPORT and FLIGHT to NOS/VE permanent files $USER.AIRPORT and $USER.FLIGHT is shown in figure 12-2. The task includes commands and the FORTRAN program. Migration occurs as the program reads the files. The program reads file FLIGHT, retains the record number of the airport record to use to migrate file AIRPORT.

```
create_file_connection standard_file=$echo file=output
collect_text output=source until='**colt end**'
      program simulat (tape1,tape2)
C  Performs statistical analysis on flights.
C  Requires location of airports to calculate flight length.
C  Array LL contains latitude and longitude in degrees.
C  Array D contains the direction for latitude and longitude.
C         .
C         .
C         .
      call convert
C
      call main
C
      end

      subroutine convert
C Migrates files:  FLIGHT (SQ records of text flights)
C                  AIRPORT (WA records on air terminals)
C  Reads and migrates file FLIGHT.
C  Uses airport record numbers in file FLIGHT to read
C  file AIRPORT by storing them in IRAY.
C  Reads and migrates records in file AIRPORT.
C  Items DEPART,ARRIVE,and CODE are airport codes.
C
C  Items for file FLIGHT
      character*3 depart,arrive
      dimension iray(99)
```

(Continued on next page)

Figure 12-2. Migration Task for a Sequential and a Direct File

```
+--(Continued from previous page)-----------------------------------------------------------+
|                                                                                            |
|  C  Items in file AIRPORT                                                                  |
|        character code*3, termnl*29                                                         |
|        character city*19, state*25                                                         |
|        real ll(2)                                                                          |
|        character*1 d(2)                                                                    |
|  C                                                                                         |
|  C  read flight                                                                            |
|        print *, ' Begin migrating FLIGHT'                                                  |
|        open (2,file='flight')                                                              |
|        do 10 I=1,99                                                                        |
|     10 iray(I) = 0                                                                         |
|     20 read (2,'(a3,1x,i2,1x,a3,1x,i2)',end=30) depart,irec1,arrive,irec2                  |
|        iray(irec1)=irec1                                                                   |
|        iray(irec2)=Irec2                                                                   |
|        go to 20                                                                            |
|     30 print *, ' migrated file flight'                                                    |
|        close (2)                                                                           |
|  C                                                                                         |
|  C  Read file AIRPORT                                                                      |
|        print *, ' Begin migrating AIRPORT'                                                 |
|        open(1,status='old',file='airport',access='direct',recl=13)                         |
|        kount = 0                                                                           |
|        do 40 n=1,99                                                                        |
|        if (iray(n) .eq. n)  then                                                           |
|          kount = kount + 1                                                                 |
|          read (1,rec=n) code,termnl,city,state,ll,d                                        |
|        endif                                                                               |
|     40 continue                                                                            |
|        close (1)                                                                           |
|        print *, 'migrated ', kount, ' records in airport'                                  |
|        print *, 'leaving subroutine convert'                                               |
|        end                                                                                 |
|        subroutine main                                                                     |
|        print *, ' Control passed to Subroutine Main.'                                      |
|  C     :                                                                                   |
|        end                                                                                 |
|  **colt end**                                                                              |
|  fortran input=source errors=ftnout list=ftnout                                            |
|  "  Start FMA  "                                                                          |
|  create_file file=$user.flight                                                             |
|  set_file_attributes file=$user.flight block_type=system_specified record_type=variable    |
|  create_file file=$user.airport                                                            |
|  set_file_attributes file=$user.airport block_type=system_specified record_type=variable   |
|  open_file_migration_aid                                                                   |
|  open_170_state                                                                            |
|  execute_command command='GET,FLIGHT.'                                                    |
|  execute_command command='FILE,FLIGHT,FO=SQ,RT=Z,BT=C,FL=20.'                             |
|  execute_command command='GET,AIRPORT.'                                                   |
|  execute_command command='FILE,AIRPORT,FO=WA,RT=U,BT=C,MRL=110.'                           |
|  close_environment                                                                         |
|  execute_migration_task ..                                                                 |
|      migration_files=((flight,$user.flight),(airport,$user.airport))                       |
|      files=lgo                                                                             |
|  close_environment                                                                         |
|  delete_file_connection standard_file=$echo file=output                                    |
|                                                                                            |
+--------------------------------------------------------------------------------------------+
```

Figure 12-2. Migration Task for a Sequential and a Direct File

Commands in the task include several processing steps. CREATE_FILE_CONNECTION specifies a file connection between standard file $ECHO and file OUTPUT so that commands are displayed to the terminal as the task executes. COLLECT_TEXT writes the FORTRAN source code to file SOURCE. Other commands specify FORTRAN compilation and using FORTRAN FMA.

You can run the task by putting the commands and program as listed in a NOS/VE file. Specify execution of the file by entering the INCLUDE_FILE command. For example, to execute a file named FMAJOB2, specify:

```
/include_file  file=fmajob2
```

## FMA Execution Considerations

This discussion details results of execution, points out considerations, and describes diagnostics. The subjects are:

- Positioning sequential migration files
- Positioning FORTRAN direct access migration files
- Ensuring data migration
- Multiple file files
- Migrating boolean items
- Characteristics of migrated NOS/VE files

### Positioning Sequential Migration Files

To help ensure error free, complete file migration, FMA performs positioning operations on sequential files involved in migration processing. (Migration processing occurs on files specified in the MIGRATE_FILES parameter of the EXECUTE_MIGRATION_TASK command.)

Positioning operations that occur when the source file is opened for migration are:

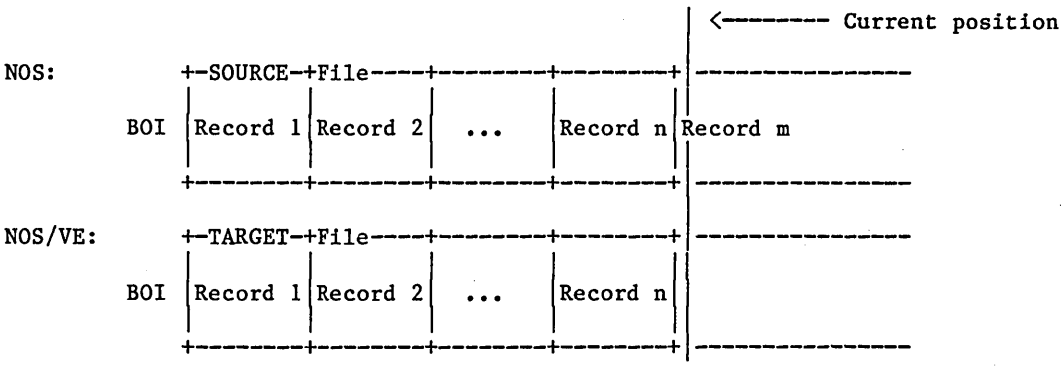
- Rewinding the source file.
- Truncating the target file to zero length. This results in any data in the target file being lost.

If the program closes and reopens the source file after migration, FMA interprets the close as ending migration and the reopen as starting migration. On the reopen, FMA would perform the two operations listed above. To complete file migration in this situation, the program would have to reread the entire source file.

After the program completes migrating a file and closes it (either explicitly or by ending execution), FMA closes and rewinds both the source and target files. Your files complete migration positioned at beginning-of-information (BOI).

To protect data in the target file once migration processing begins, FMA does not REWIND (or BACKSPACE) the target file when the program specifies these operations for the source file. FMA holds the target file at the current position, remembers the number of records repositioned, and resumes writing to the target file only after the program has reread all the records repositioned.

Assume file SOURCE is a CYBER 170 file being migrated to NOS/VE file TARGET. The FORTRAN program has read n records:




FORTRAN statements and effect on file position:

```
REWIND (SOURCE)  <-- Positions SOURCE at BOI.  Leaves TARGET at end.
READ source ...  <-- Required n times before SOURCE and TARGET are positioned at
                     corresponding points.
```

The program must reread n records before FMA resumes migrating records (writing records to TARGET). With the next read, FMA migrates record m.

### Positioning FORTRAN Direct Access Migration Files

To help ensure error free, complete file migration, FMA performs positioning operations on FORTRAN direct access files involved in migration processing. (Migration processing occurs on files specified in the MIGRATE_FILES parameter of the EXECUTE_MIGRATION_TASK command.)

When the FORTRAN program opens the source file, FMA begins migration processing by:

- Positioning the source file at beginning-of-information.
- Truncating the target file to zero length. This results in any data in the target file being lost.

If the program closes and reopens the source file after migration, FMA interprets the close as ending migration and the reopen as starting migration. On the reopen, FMA would perform the two operations listed above. To complete file migration in this situation, the program would have to reread the entire source file.

### Ensuring Data Migration

For FMA to migrate data, the FORTRAN program must read the data. The following rules apply:

For unformatted files (either sequential or direct access), only data that the program reads is migrated. If the program reads a partial record, FMA migrates only the part of the record that is read.

For formatted files (either sequential or direct access) and for list directed and NAMELIST sequential files, FMA migrates a complete record if a part of the record is read.

To completely migrate sequential files, the program must read each record in the file.

FMA issues diagnostics to warn of incomplete file migration:

When the program fails to read all records sequential files.

When the program reads partial records in unformatted sequential files.

FMA does not indicate incomplete migration of direct access files.

If you attempt to migrate a file twice (as you might if the first attempt results in an incomplete migration), FMA deletes the original file and creates a new one.

For an incomplete migration where the program fails to read all of the records, you can retain the migrated file (by making it permanent and changing its name) and then run the program again to migrate the rest of the records. You can then use Sort/Merge to merge the first file with the second file.

### Multiple File Files

A common practice in using multiple file files is not supported by FORTRAN FMA. Multiple file files that have records resulting from different input/output processing cannot be completely migrated by FMA. You may have to study your application program to ensure that this is not a problem.

Ensure that you need only one FILE command to describe the multiple-file file.

Ensure that your program that migrates the file does not open, close, and reopen the source file. The second open causes the target file to be truncated to zero length so that all data in it is lost.

### Migrating Boolean Items

Data of type boolean is migrated as a bit pattern. In migrating boolean data, FMA issues a nonfatal diagnostic in the following situations:

Unformatted reads containing boolean variables in I/O lists

Formatted reads with boolean edit descriptors

List-directed and NAMELIST reads with boolean constants in input

When FMA issues a nonfatal error for use of boolean variables, FMA continues processing the remainder of the I/O list (unless the program provides an ERR= exit or an IOSTAT= variable).

For unformatted reads, FMA stores the 60-bit CYBER 170 word bit pattern in the receiving 64-bit CYBER 180 word, right justified and zero filled.

It is sometimes possible to convert boolean files by changing the type BOOLEAN specification for each word to the correct type specification for the conversion, such as REAL, INTEGER, and so forth. For example, if your boolean data really has integer values, change the data specification statements and read statements to read the data as integers.

Hollerith data should be specified as type CHARACTER.

You might find it necessary to do some processing before conversion to set up the file for conversion. Alternatively, you might find it necessary to do processing after conversion to perform the final conversion for the application.

### Characteristics of Migrated NOS/VE Files

When a file is migrated by FMA, the default file attributes establish the characteristics of the file. The attributes of the target file are defined by NOS/VE and FORTRAN depending on the type of input/output creating the file.

After a file is created (that is, after the file is opened for the first time), you can change the preserved file attributes that define the structure of the file. The attributes that can be overridden by a SET_FILE_ATTRIBUTE (SETFA) command or CHANGE_FILE_ATTRIBUTE (CHAFA) command prior to file creation are indicated by an asterisk. The attributes that can be overridden prior to any open of the file are indicated by two asterisks. Connecting files to $INPUT and $OUTPUT affects file characteristics. Do not connect any file being migrated to either $INPUT or $OUTPUT.

Tables 12-1 and 12-2 give the default attributes of migrated files. For more information about NOS/VE files, see chapter 10, File Interface Introduction.

Table 12-1. Defaults for File Attributes on NOS/VE for Sequential I/O

| File Attribute | Formatted,List Directed NAMELIST, Sequential I/O | Unformatted Sequential I/O |
|---|---|---|
| MAXIMUM_RECORD_LENGTH | RECL= in OPEN or FILE command | RECL= in OPEN or FILE command |
| OPEN_POSITION | $BOI† | $BOI† |
| ACCESS_MODE | R/W/A/M† | R/W/A/M† |
| FILE_ORGANIZATION | SQ | SQ |
| RECORD_TYPE | V†† | V†† |
| PAGE_WIDTH | 132 | not applicable |

†Can be overridden by SET_FILE_ATTRIBUTES command prior to any open.
††Can be overridden by SET_FILE_ATTRIBUTES command prior to file creation.

| | |
|---|---|
| $BOI | Beginning of information. |
| R/W/A/M | READ/WRITE/APPEND/MODIFY. |
| SQ, BA | Sequential, byte addressable. |
| V, F, U | Variable-length, fixed-length, undefined. |

Table 12-2. Defaults for File Attributes on NOS/VE for Direct I/O

| File Attribute | Direct Access I/O |
|---|---|
| MAXIMUM_RECORD_LENGTH | RECL= in OPEN or FILE command |
| OPEN_POSITION | not applicable |
| ACCESS_MODE | R/W/A/M† |
| FILE_ORGANIZATION | BA |
| RECORD_TYPE | F†† |
| PADDING_CHARACTER | blank |
| PAGE_WIDTH | not applicable |

†Can be overridden by SET_FILE_ATTRIBUTES command prior to any open.
††Can be overridden by SET_FILE_ATTRIBUTES command prior to file creation.

R/W/A/M  READ/WRITE/APPEND/MODIFY.
SQ, BA  Sequential, byte addressable.
V, F, U  Variable-length, fixed-length, undefined.

## COBOL File Migration Aid

The COBOL File Migration Aid (COBOL FMA) provides a COBOL-oriented method for migrating COBOL 5 files from NOS to NOS/VE. The description of COBOL FMA is divided into the following topics:

Overview

Data Format Conversion

Handling Symbolic Keys

Executing the 170 FILE Command

Steps in Executing COBOL FMA

Summary of Steps in Executing COBOL FMA

COBOL FMA Command Descriptions

COBOL FMA Examples

### COBOL FMA Overview

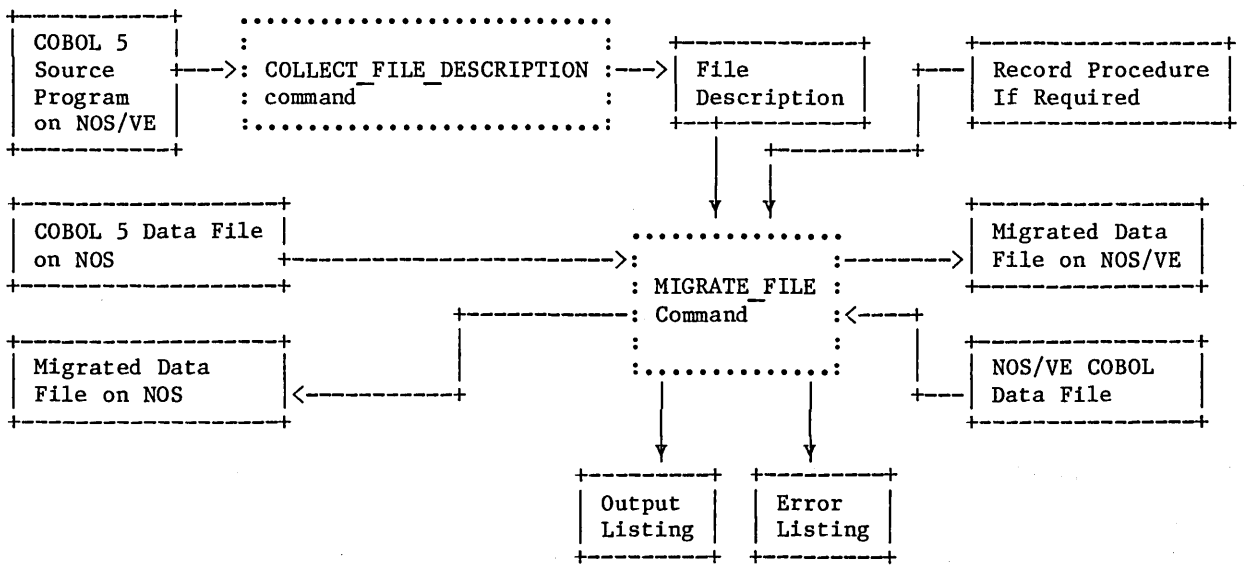
COBOL FMA provides a COBOL-oriented method for migrating COBOL files between NOS and NOS/VE. To migrate a COBOL file in either direction, you must describe the structure of that file to COBOL FMA in COBOL 5 terms. COBOL FMA provides you with a command that simplifies this process; the command extracts the file description from an existing COBOL 5 source program.

In some cases, you must provide additional information to resolve ambiguities in the record description of the file. For example, if your COBOL 5 program uses multiple 01 descriptors or redefines items in a record, you must specify how the record is to be processed. COBOL FMA allows you to write a COBOL-like procedure to resolve these ambiguities.

As COBOL FMA migrates your COBOL file, it produces a listing that indicates how specific computational data items in the record description must be changed to be used in a NOS/VE COBOL program that accesses the migrated file.

The capability of migrating a NOS/VE COBOL file to NOS (reverse migration) is useful in cases where output from a migrated program is used as input to a program that has not been migrated.

The following diagram illustrates some of the key components in COBOL file migration and the relationships between them:




The COBOL FMA commands that you use to migrate a file are:

OPEN_FILE_MIGRATION_AID

Enters the FMA utility so that you can enter FMA commands.

COLLECT_FILE_DESCRIPTION

Extracts the file description from a COBOL 5 source program.

MIGRATE_FILE

Causes FMA to read the file description, any record procedure, and the COBOL 5 file to be migrated, and to output the migrated file along with a listing.

(COBOL FMA also provides the OPEN_170_STATE command and the EXECUTE_COMMAND command that allow you to execute a NOS command from within the FMA utility.) All of these commands are described in more detail later.

## Data Format Conversion

When you migrate a COBOL data file from NOS to NOS/VE or from NOS/VE to NOS, FMA converts COBOL data formats to the formats of the target system. The COBOL 5 reference manual and the COBOL Usage manual describe the data formats for each COBOL dataname type.

A file that is to be migrated from NOS/VE to NOS must have been created by a valid COBOL 5 program compiled under NOS/VE with BASE_LANGUAGE=COBOL5 specified on the COBOL command. Typically, this is a program that was previously migrated from NOS to NOS/VE. Thus, the data types used in the program must be valid COBOL 5 data types.

FMA resolves differences in computational data type formats as follows:

| COBOL 5 | NOS/VE COBOL |
|---|---|
| COMP <----------> | DISPLAY |
| Display code numeric | ASCII numeric |
| COMP-1 <----------> | COMP<br>BINARY |
| Numeric class<br>Uses full 60-bit word<br>Stored as 48-bit binary integer<br>Right-aligned<br>FORTRAN integer type | Numeric class<br>Decimal numeric value<br>Stored as binary integer<br>Size depends on PIC clause<br>FORTRAN integer type (if size=18 and synchronized) |
| COMP-2 <----------> | COMP-1 |
| Single-precision floating-point<br>Uses full 60-bit word<br>No PICTURE clause<br>Value is signed normalized floating-point number<br>Up to 14 significant digits<br>FORTRAN real type | Single-precision floating-point<br>Uses full 64-bit word<br>No PICTURE clause<br>Value is signed normalized floating-point number<br>Up to 14 significant digits<br>FORTRAN real type |
| (COBOL 5 does not provide a data type equivalent to the NOS/VE COMP-2 data type.) | COMP-2<br>Double-precision floating-point number<br>Up to 28 significant digits |
| COMP-4 ----------> | COMP<br>BINARY (described above) |
| Numeric class<br>Size depends on PIC clause<br>Binary integer<br>Maximum of 48 bits<br>Signed or unsigned | COMP-3<br>PACKED DECIMAL<br>Numeric class<br>String of 4-bit representations of numeric digits packed two per byte with optional sign represented as rightmost 4 bits in rightmost byte. In some cases, this is the best conversion for a COMP-4 item. |

COBOL FMA converts COBOL 5 COMP-1 items to COBOL-for-NOS/VE COMP items. However, to use the migrated data file successfully, you must specify BASE_LANGUAGE=COBOL5 on the NOS/VE COBOL command when you compile your COBOL-for-NOS/VE program. You can compile with the BASE_LANGUAGE parameter not set to COBOL5 if you first make the following changes to the file description in your COBOL source program:

- Change the data item to SYNCHRONIZED.
- Ensure that the data item has enough digits to fill a complete NOS/VE 64-bit word.

For example:

```
PIC 9(18) COMP SYNCHRONIZED
PIC 9(14)V9(4) COMP SYNCHRONIZED
```

If an item does not have data in a format compatible with its definition, a warning message is issued and a default value compatible with the converted item definition is used in the resulting record. The following table shows the defaults used.

| INPUT FILE DATA TYPE | CORRECTED OUTPUT VALUE |
|---|---|
| display | all blanks; all zeros if numeric |
| computational | all zeros |
| computational-1 | binary zero |
| computational-2 | single-precision floating-point zero |
| computational-4 | binary zero |

## Handling Symbolic Keys

If the COBOL 5 data file that you are migrating contains symbolic keys, you must decide whether you want to retain the ordering of records in the file.

You can migrate a COBOL 5 data file containing symbolic keys without doing anything special. However, symbolic keys will be treated as uncollated keys; therefore, the ordering of records will not be retained. The records will be reordered according to the ASCII collating sequence.

IMPORTANT

To handle symbolic keys as described above, you must not specify BASE_LANGUAGE=COBOL5 on the COBOL command when you compile your NOS/VE COBOL program.

To retain the record ordering, you must specify that the key type of the symbolic keys is collated. You can also specify the name of the collating sequence to be used to determine record ordering. You do this using the SET_FILE_ATTRIBUTES command on NOS/VE; for example:

```
/set_file_attributes  file=$user.myfile ..
.. /key_type=collated ..
.. /collate_table_name=collating_table_name
```

You can enter this SET_FILE_ATTRIBUTES command either before or after entering COBOL FMA.

You can omit the COLLATE_TABLE_NAME parameter; if you do, the COBOL6_FOLDED collating sequence is used.

IMPORTANT

If you use the SET_FILE_ATTRIBUTES command as described above, you must specify BASE_LANGUAGE=COBOL5 on the COBOL command when you compile your NOS/VE COBOL program.

The SET_FILE_ATTRIBUTES command affects only symbolic keys; it has no effect on integer keys.

NOTE

When migrating a file with alternate keys from NOS/VE to NOS, the sum of the lengths of the primary key and alternate key must not exceed 230 characters.

## Executing the 170 File Command

The OPEN_170_STATE and EXECUTE_COMMAND commands enable you to execute a NOS command from within COBOL FMA. (These commands are described later.)

You might need to use these commands to specify the block type (BT) of the COBOL 5 data file. The default block type depends on the device; for S or L tapes, the default depends on the BLOCK CONTAINS clause in the input file description (K or E blocks). If the device is not an S or L tape, the default is C.

You can override this default by executing a 170 FILE command; for example:

```
/open_170_state
/  execute_command  command='FILE,MYFILE,BT=x.'
/close_environment
```

You might also need to use these commands to specify the record mark character if the file you are migrating contains R-type records. You need to do this only if your COBOL 5 program contains a USE clause that specifies a record mark character other than the Cyber Record Manager default.

Simply execute a 170 FILE command, specifying the RMK parameter to be the same character as specified in the COBOL 5 USE clause.

If you change the file organization when migrating a file from NOS/VE to NOS, be sure that you specify all of the NOS file attributes on the FILE command. If you do not do this, defaults are used for the unspecified attributes, which may lead to unexpected results caused by overlapping FIT fields.

## Steps in Executing COBOL FMA

To migrate a file using COBOL FMA, follow these steps:

1. Move your COBOL source program from NOS to NOS/VE. To do this, log in to NOS/VE and use the GET_FILE command. For example:

   ```
   /get_file  to=cobprog  data_conversion=d64
   ```

2. For NOS-to-NOS/VE migration, create a NOS/VE file to receive the migrated version of your COBOL data file. To do this, use the CREATE_FILE command. For example:

   ```
   /create_file  file=master
   ```

   To ensure correct values for the record type and block type of each output NOS/VE migrated file, use a SET_FILE_ATTRIBUTES command:

   ```
   SET_FILE_ATTRIBUTES  FILE=output file ..
      BLOCK_TYPE=keyword ..
      RECORD_TYPE=keyword
   ```

   For more information about which values to use for BLOCK_TYPE and RECORD_TYPE, see the CYBIL File Interface Usage manual or the COBOL Usage manual.

3. If the input file description for the file you are migrating contains multiple 01 file descriptions or uses a REDEFINES clause, you must provide a record procedure to process file description ambiguities. (A record procedure is not required if your COBOL 5 program contains multiple 01 descriptions for some other file or within the working storage section.) Create a record procedure using COLLECT_TEXT or the Full-Screen editor; record procedure commands are described later.

4. Start COBOL FMA utility using the OPEN_FILE_MIGRATION_AID command.

5. Optionally, extract the file description from your COBOL 5 source program using the COLLECT_FILE_DESCRIPTION command; for example:

```
FA/collect_file_description input=cobprog ..
                            output=description ..
                            assigned_name=master
```

This gets the file description from the COBOL 5 program on file COBPROG and places it on file DESCRIPTION.

6. Next, for NOS-to-NOS/VE migration, ensure that your COBOL 5 data file is a NOS local file. You can issue NOS commands by opening 170 state and then using the EXECUTE_COMMAND command. For example:

```
FA/open_170_state
FA/execute_command  command='ATTACH,MASTER,OM12345.'
FA/close_environment
```

In this example, if a file by the name of MASTER is not local to the 170 job, FMA attempts to ATTACH the file; if the ATTACH fails, the MIGRATE_FILE command will terminate abnormally. You might also need to execute the FILE command while the 170 state is open if the file attributes of your COBOL5 program do not accurately describe the file.

After issuing any necessary NOS commands, close the 170 state environment using the CLOSE_ENVIRONMENT command as shown above.

7. For NOS/VE-to-NOS migration, be sure that a permanent file having the same name as the migrated file does not already exist. It it does, FMA will not purge the file; it will issue an error message and abort. If a local file having the same name exists, FMA will write that file. If a local or permanent file having the same name does not exist, FMA will create a new direct access permanent file.

You can purge an existing permanent file by specifying an EXECUTE_COMMAND command that executes a PURGE command. For example:

```
FA/open_170_state
FA/execute_command  command='PURGE,MASTER.'
FA/close_environment
```

8. Now migrate the file using the MIGRATE_FILE command. For example:

```
FA/migrate_file input=master ..
                input_file_description=description ..
                record_procedure=process ..
                output=$user.master
```

This command migrates a file from NOS to NOS/VE. The migrated file is placed on file $USER.MASTER. The INPUT and OUTPUT parameters are optional; if omitted, MIGRATE_FILE uses the ASSIGNED_NAME specified in the file description.

9. Finally, leave COBOL FMA by issuing the CLOSE_ENVIRONMENT command.

## Summary of Steps in Executing COBOL FMA

1. Use GET_FILE to move your COBOL source program from NOS to NOS/VE.
2. For NOS-to-NOS/VE migration, create a NOS/VE file (CREATE_FILE) to receive the migrated version of your data file. Enter a SET_FILE_ATTRIBUTES command to ensure the correct values for the NOS/VE migrated file for block type and record type.
3. Use COLLECT_TEXT or Full-Screen editor to write any needed record procedure.
4. Initiate the COBOL FMA utility with OPEN_FILE_MIGRATION_AID.
5. Optionally, use COLLECT_FILE_DESCRIPTION to extract the file description from your COBOL 5 source program.
6. If needed, issue any NOS commands by opening the 170 state (OPEN_170_STATE), executing NOS commands (EXECUTE_COMMAND), and then closing 170 state (CLOSE_ENVIRONMENT).
7. Use the MIGRATE_FILE command to migrate the file.
8. Use the CLOSE_ENVIRONMENT command to leave COBOL FMA.

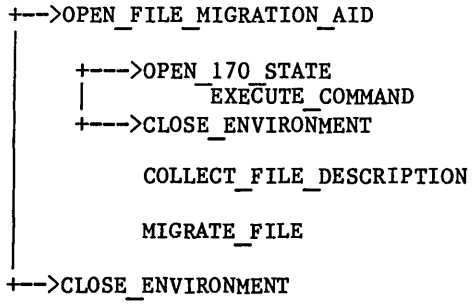
## COBOL FMA Command Descriptions

Following are descriptions of each of the COBOL FMA commands. Note that the command formats show each parameter on a separate line without the ellipsis required for continuation in executable code. The syntax of COBOL FMA commands follows the System Command Language (SCL) conventions. The commands are:



### Overview of COBOL FMA Commands

COBOL FMA commands must be entered in a particular order because the commands provide a processing path through a migration environment. Each command either opens a specialized processing environment, executes an operation in the environment, or closes an environment. The following diagram indicates the execution order; commands can be executed only within the containing bracket.



### OPEN_FILE_MIGRATION_AID (OPEFMA)

The OPEN_FILE_MIGRATION_AID (OPEFMA) command invokes FMA and opens the file migration environment. You must execute this command to use any other COBOL FMA command.

Format:
```
OPEN_FILE_MIGRATION_AID
     PARTNER_JOB_CARD=string
     STATUS=variable
```

PARTNER_JOB_CARD — The job card for the CYBER 170 partner job; optional. The partner job executes as a batch job. The value of this parameter must correspond to NOS job card syntax.

Any job card parameters you specify here remain in effect for the duration of the FMA utility task. If the partner job exceeds the limits you specify with this parameter, FMA terminates abnormally.

If you do not specify this parameter, a default job card is used. If the NOS/VE job is a batch job, no parameters are specified on the default partner job card. If the NOS/VE job is interactive, the default partner job card specifies an infinite time limit; no other job card parameters are specified.

STATUS — Status variable; optional.

Example:
```
/open_file_migration_aid
FA/
```
or
```
/opefma
FA/
```

### OPEN_170_STATE (OPE1S)

The OPEN_170_STATE command opens access to the CYBER 170 and allows you to execute NOS commands.

Format:
```
OPEN_170_STATE
     STATUS=variable
```

Parameter: Status variable; optional.

Example:
```
FA/open_170_state
```
or
```
FA/ope1s
```

### EXECUTE_COMMAND (EXEC)

The EXECUTE_COMMAND command executes a NOS command from within FMA. (To use this command, you must first open a connection to the CYBER 170 by issuing the OPEN_170_STATE command.)

Any diagnostics issued by the NOS commands are written to the NOS/VE job log. Thus, if a fatal error occurs, you can determine which NOS command caused the error by examining the job log. (Use the DISPLAY_LOG command to display the job log.)

Load sequences are processed as single commands; therefore, the entire load sequence must be entered on a single EXECUTE_COMMAND command.

Because FMA uses NOS local files internally, you should use caution when manipulating NOS local files through EXECUTE_COMMAND. In particular, never return all local files (RETURN,* or CLEAR) and do not use an EXIT control card.

Format:

```
EXECUTE_COMMAND
     COMMAND='nos-command.'
     STATUS=variable
```

COMMAND — A NOS command entered as a string, terminated with a period, and enclosed in apostrophes. Maximum string length is 80 characters. For example, attaching file AFILE:

```
FA/execute_command  command='ATTACH,AFILE.'
```

Optionally, NOS commands can appear as a list of strings. NOS received each string as a separate line. For example:

```
FA/execute_command  command=('LDSET=OLIB.' ..
                             'LGO.')
```

If this command is used in an SCL procedure, SCL can perform parameter substitution. The command is passed to NOS after substitutions occur.

STATUS — Status variable; optional.

Examples: Specifying commands with the parameter name:

```
FA/execute_command  command='ATTACH,DATAFIL.'
FA/exec  c='FILE,DATAFIL,FO=SQ,RT=Z,BT=C,EO=A,FL=2000.'
```

Specifying commands by position:

```
FA/execute_command  'ATTACH,DATAFIL.'
FA/exec  'FILE,DATAFIL,FO=SQ,RT=Z,BT=C,EO=A,FL=2000.'
```

## COLLECT_FILE_DESCRIPTION (COLFD)

The COLLECT_FILE_DESCRIPTION command extracts a file description from a COBOL 5 source program and places the description on the file you specify.

Format:

```
COLLECT_FILE_DESCRIPTION INPUT=source-file
                         OUTPUT=description-file
                         FD_NAME=fd-entry
                         ASSIGNED_NAME=assigned-name
                         RECORD_NAME=list-of-01-names
                         LIST=listing-file
                         ERROR=error-file
                         STATUS=variable
```

INPUT — Specifies the NOS/VE file containing your COBOL 5 source program. The file must contain only one program; that program must be complete and correct. (Any COPY and REPLACE statements in the program are not honored by COLLECT_FILE_DESCRIPTION.)

OUTPUT — Specifies the file to receive the selected input file description.

FD_NAME — Specifies the name of the FD entry to be collected from the source program; optional. This parameter must not be specified if the ASSIGNED_NAME parameter is specified.

| | |
|---|---|
| ASSIGNED_NAME | Specifies the assigned name in the SELECT clause of the FILE-CONTROL paragraph for the file for which the collection is to be performed; optional. This parameter must not be specified if the FD_NAME parameter is specified. |
| RECORD_NAME | Specifies a list of 01 names that are not members of any FD entry; optional. The named 01 entries and their subordinate items are placed on the output file in the order specified by this parameter. To specify RECORD_NAME, you must also specify the FD_NAME or ASSIGNED_NAME parameter. |
| LIST | Specifies the file to receive a source listing of the COBOL 5 program; optional. The default is $LIST. |
| ERROR | Specifies the file to receive error diagnostics; optional. The default is $ERRORS. |
| STATUS | Status variable; optional. |

FD_NAME and ASSIGNED_NAME parameters are required in batch mode, but optional in interactive mode; if both are omitted in interactive mode:

COLLECT_FILE_DESCRIPTION displays a list of FD names, assigned names, and 01 names that are not members of any FD.

You can then select any members of the displayed list to be on the output file; however, you must select one file and you can optionally select one or more 01 names.

If your data file has a nonembedded key, COLLECT_FILE_DESCRIPTION creates a 77 level item for the dataname specified on the record key clause. The dataname retains the data definition specified in the original source program. The 77 item is placed in the working-storage section of the resulting file description.

Example:
```
FA/collect_file_description input=cobprog ..
                            output=description ..
                            assigned_name=master
```
or
```
FA/colfd i=cobprog ..
         o=description ..
         an=master
```

### MIGRATE_FILE (MIGF)

The MIGRATE_FILE command migrates a COBOL data file from the NOS to NOS/VE or from NOS/VE to NOS. This command migrates the file as follows:

A. MIGRATE_FILE checks the input file description created by COLLECT_FILE_DESCRIPTION for errors; any computational data type changes are written to the list file.

B. If a record procedure is defined for the file, that procedure is checked for errors.

C. Each record of the data file is migrated by repeating the following:

   1. The next record is read from the CYBER 170 file.

   2. Unambiguous input file descriptions are used to convert the parts of the record they describe. Ambiguous input file descriptions are processed as follows:

      a. If the input file description contains multiple 01's, the record procedure is executed immediately; see step 2.c.

b. All fields are converted except those that have overlapping definitions (via REDEFINES).

c. The record procedure is executed. For each conversion reference executed, step 2.b above is performed.

3. The converted record is written to the target file.

Format:

```
MIGRATE_FILE INPUT=input file
             OUTPUT=(output-file,index-file)
             MIGRATION_NAME=name
             INPUT_FILE_DESCRIPTION=(description-file,keyword)
             RECORD_PROCEDURE=procedure-file
             LIST=list-file
             ERROR=error-file
             STATUS=variable
```

INPUT — Optional. Specifies the name of the file to be migrated. For NOS-to-NOS/VE migration, the file must be a NOS local file. For NOS/VE-to-NOS migration, this parameter can specify a file path. The default is the name in the ASSIGN clause of the input file description.

OUTPUT — Optional. The first value is the name of the file to receive the migrated version of the input file. For NOS-to-NOS/VE migration, file-name can be a file path. For NOS/VE-to-NOS migration, file-name is a NOS file name; FMA creates the file as a direct access permanent file. If a permanent file having the specified name already exists, a fatal error occurs. (FMA does not attempt to purge the file.) If a local file having the specified name exists, FMA writes to that file.

The default is name in the ASSIGN clause of the input file description.

For indexed sequential files with alternate keys only, the second value is the name of a NOS file to receive the index. FMA creates a direct access permanent file having the specified name. The default is taken from the ASSIGN clause of the input file description.

MIGRATION_NAME — Reserved for future use.

INPUT_FILE_DESCRIPTION — Specifies the name of the file containing the input file description for the COBOL 5 data file to be migrated. For migration in either direction, this file must contain a complete and correct COBOL 5 source program that has only one SELECT clause. The file named here can be the file named in the OUTPUT parameter of a previous COLLECT_FILE_DESCRIPTION command.

The second value is an optional keyword that specifies the direction of the migration:

| | |
|---|---|
| C170_COBOL | NOS to NOS/VE (default) |
| C180_COBOL_CC | NOS/VE to NOS |

RECORD_PROCEDURE — Specifies the name of the file containing a record procedure. A record procedure is required if the input file description contains multiple 01's or a REDEFINES clause.

LIST — Specifies the file to receive a source listing; optional. The listing contains an input file description listing and a listing of the record procedure if a record procedure is used. The default is $LIST.

ERROR — Specifies the file to receive error diagnostics; optional. The default is $ERRORS.

STATUS — Status variable; optional.

Example: The following example migrates a file from NOS to NOS/VE:

```
FA/collect_file_description input=cobprog ..
                            output=description ..
                            assigned_name=master
FA/migrate_file  input=cobdata ..
                 output=$user.my_great_file ..
                 input_file_description=description
```

or

```
FA/colfd i=cobprog ..
         o=description ..
         an=master
FA/migf i=cobdata ..
        o=$user.my_great_file ..
        ifd=description
```

The following example migrates a file from NOS/VE to NOS:

```
FA/collect_file_description input=myprog ..
                            output=mydesc ..
                            assigned_name=master

FA/migrate_file input=cobfile ..
                output=nosfile ..
                input_file_description=(mydesc, c180_cobol_cc)
```

## CLOSE_ENVIRONMENT (CLOE)

The CLOSE_ENVIRONMENT (CLOE or QUIT) command closes the current environment; subsequent commands are executed in the containing environment.

Format:

```
CLOSE_ENVIRONMENT
            STATUS=variable
```

Parameter: Status variable; optional.

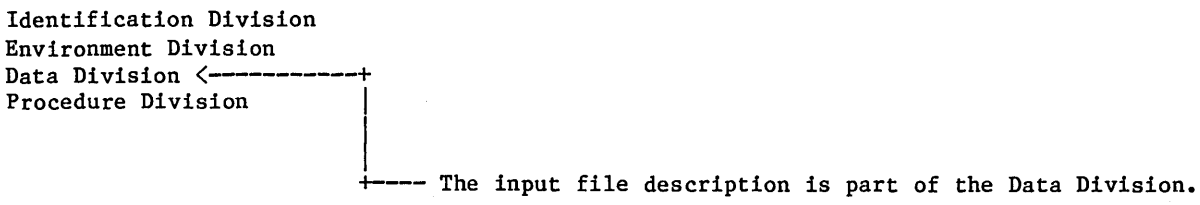
Example:

```
/open_file_migration_aid
FA/open_170_state
.
.
FA/close_environment
.
.
FA/close_environment
/
```

The first CLOSE_ENVIRONMENT closes the CYBER 170 connection, but leaves the FMA utility active; the second CLOSE_ENVIRONMENT terminates FMA and returns to NOS/VE.

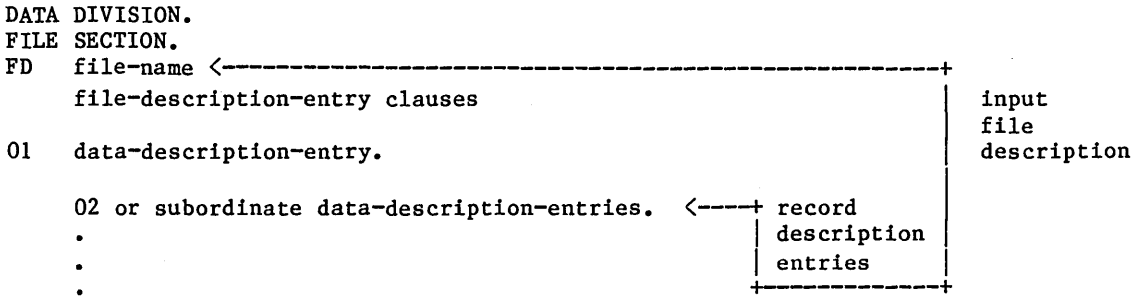
## Input File Description Overview

An input file description is the portion of your COBOL 5 program that describes the COBOL 5 data file to be migrated. Recall that a COBOL program consists of the following divisions:




One section within the Data Division is the File Section. The File Section contains input file descriptions (which are denoted by FD). Within a file description are data description entries (denoted by 01), and within a data description entry, there can be record description entries.

For example:




The MIGRATE_FILE command uses this input file description as input. The COLLECT_FILE_DESCRIPTION command extracts the input file description from your COBOL 5 source program.

## Record Procedure Command Descriptions

A record procedure is a series of statements that specify how particular records of your COBOL 5 data file are to be migrated. A record procedure is required only when multiple 01 descriptors or REDEFINES clauses exist in the record description.

Overview of Record Procedures
Conversion References
IF Constructs

### Overview of Record Procedures

A record procedure is a series of statements that specify how particular records of your COBOL 5 data file are to be migrated. A record procedure is required only when the input file description contains multiple 01's or a REDEFINES clause.

For example, a record procedure is required to migrate the data file of the following COBOL 5 program because the record description in the program contains more than one 01 descriptor:

```
FILE SECTION.
FD SALES_FILE.
.
.
.
01 SALES-RECORD.
.
.
.
01 PROFITS-RECORD.
```

A record procedure is required to migrate the data file of the following COBOL 5 program because the record description in the program contains a REDEFINES clause:

```
01 EMPLOYEE-RECORD.
.
.
.
    02 EXEMPT-DATA.
    .
    .
    .
    02 NON-EXEMPT-DATA REDEFINES EXEMPT-DATA.
```

That is, because the NON-EXEMPT-DATA structure redefines the EXEMPT-DATA structure, you must specify to FMA which structure is to be used to migrate each particular record. This is the purpose of a record procedure.

A record procedure is written in a language similar to COBOL; however, a record procedure contains only two types of statements:

Conversion references, which cause the content of datanames in an input record to be migrated to the output record

IF constructs, which select a block of statements to execute based on conditions you specify

The following is an example of a record procedure:

```
IF GRADE IS GREATER THAN 9
     THEN EXEMPT-DATA
     ELSE NON-EXEMPT-DATA
ENDIF
```

These four lines comprise an IF construct that executes one of two conversion references depending on the value of GRADE. Conversion references and IF constructs are described later.

The elements you use to write a record procedure are:

Datanames (names in the record description); datanames can be qualified in the normal COBOL style; nonunique datanames must be qualified. Level 66 items cannot be referenced in a record procedure.

Literals; nonnumeric literals are delimited by apostrophes or quotes and numeric literals are specified using standard COBOL conventions.

Intrinsic operands; these are the keywords $NUMERIC, NUMERIC, $INTEGER, INTEGER, and $RECORD-LENGTH.

Separators are any number of spaces.

### Conversion References

You use a conversion reference to migrate the content of a dataname in the input record to its corresponding location in the output record.

You represent a conversion reference by specifying the dataname of the item to be migrated. For example:

```
EMPLOYEE-RECORD
```

causes item EMPLOYEE-RECORD to be migrated.

The dataname can be qualified using COBOL conventions, and must be qualified if the dataname is not unique.

The dataname you specify can be an elementary item or a group item.

Specifying a group item is the same as specifying each of the elementary items in that group. Specifying a group item migrates all elementary items in that group; this uses the MOVE CORRESPONDING concept rather than the group move concept.

All members of an 01 must be assigned implicitly by being a member of a group or explicitly by name.

For a REDEFINES clause, one of the alternative items must be explicitly assigned. You can use an IF construct to choose one of the alternative items based on conditions you specify.

All assignments executed for a given record must belong to a single 01; if this is not the case, no diagnostic is issued.

Elementary items named FILLER are moved. For items containing or subordinate to OCCURS, all occurrences are moved.

If the data of an item is not compatible with its definition, a warning is issued and the item is set to a default value. When this occurs for a group, migration continues with the next item in the group. If the error is for an element of an OCCURS clause and the OCCURS clause does not contain a group, elements to the right of the erroneous element of the OCCURS clause are migrated; default values are supplied for each erroneous element.

### IF Constructs

You use an IF construct to test for a specified condition and execute one of several blocks of statements based on the result. An IF construct appears as follows:

```
IF expression THEN
   statement-list
ELSEIF expression THEN }
     statement-list    } (Optional and more than one can be specified.)
.
.
.
ELSE expression THEN }
     statement-list  } (Optional; if specified, must be last.)
ENDIF
```

Notice that this is similar to the IF verb in COBOL 5; however, for a record procedure, the ENDIF is not optional. (ELSEIF can also be written as IFELSE, ELSE-IF, and IF-ELSE; ENDIF can also be written as IFEND, END-IF, and IF-END.)

The boolean expression in the IF construct can contain:

```
Datanames
Literals
Intrinsic operands
```

These entities are compared using operators. The following screens list the operators that can be used.

```
AND
OR
NOT

     EQUALS                    NOT EQUAL
[IS] EQUAL [TO]          [IS] NOT EQUAL [TO]
[IS] =                   [IS] NOT = [TO]
                         [IS] <> [TO]

     EXCEEDS             [IS] NOT GREATER [THAN]
[IS] GREATER [THAN]      [IS] NOT > [THAN]
[IS] > [THAN]            [IS] <= [TO]

[IS] LESS [THAN]         [IS] NOT LESS [THAN]
[IS] < [THAN]            [IS] NOT < [THAN]
                         [IS] >= [TO]
```

Parentheses can be used to control order of evaluation.

You can use intrinsic operands in the boolean expression of an IF construct to test certain attributes of a dataname or literal. The intrinsic operands are:

| | |
|---|---|
| $NUMERIC or NUMERIC | (Used to determine if an item is numeric or not) |
| $INTEGER or INTEGER | (Used to determine if an item is integer or not) |
| $RECORD-LENGTH | (Contains the record length of the original record as it exists on the 170) |

The intrinsic operands are described in the following paragraphs.

You use the $NUMERIC or NUMERIC intrinsic operand to determine whether the value of a dataname is numeric.

For example, in the statement:

```
IF EMPLOYEE-NUMBER IS NUMERIC
```

the result of the boolean expression is TRUE if the value of the dataname conforms to the COBOL concept of CLASS NUMERIC; otherwise, the result is false.

As in the example, you can do comparisons with this intrinsic operand by using the IS operator only.

You use the $INTEGER or INTEGER intrinsic operand to determine whether the value of a dataname is an integer.

For example, in the statement:

```
IF EMPLOYEE-NUMBER IS INTEGER
```

the result of the boolean expression is TRUE if the leftmost 12 bits of the dataname value are all zero or all one; otherwise, the result is false.

The operand being compared to this intrinsic operand must be contained in a 60-bit word. You can do the comparison by using the IS operator only, as in the example.

You use the $RECORD-LENGTH intrinsic operand to return the length in characters of the original record as it exists on the 170.

You can compare the value of a dataname or a literal to the value returned by $RECORD-LENGTH.

When migrating a file with a variable record length (NOS record types D, T, W, and R) you must be certain that the item referenced by an IF is not beyond the end of the current record. You can do this by checking the $RECORD_LENGTH function, the DEPENDING field, or some other indicator known to be within the record before executing the IF.

If an IF references an item that is beyond the end of the current record, a fatal error diagnostic is issued and migration immediately stops.

Input files with Z type records are handled as fixed length records; therefore, you need not be concerned about referencing beyond the end of a Z type record.

## COBOL FMA Examples

The following examples illustrate the use of COBOL FMA. The first example migrates a file that contains no ambiguities in the file description. The second example migrates a file that does have ambiguities in the file description; the ambiguities are resolved using a record procedure. The third example migrates a file from NOS/VE to NOS.

### Simple COBOL FMA Example

This example demonstrates how to migrate a COBOL 5 data file from NOS to NOS/VE. In this example, assume that the file description for the data file to be migrated contains no ambiguities. That is, the file description in the source program contains no more than one 01 descriptor and does not use the REDEFINES clause. Because no ambiguities exist in the file description, a record procedure is not needed.

The data file to be migrated is a NOS permanent file; the file name for the data file is MASTER. MASTER is also the name assigned in the ASSIGN clause in the file description paragraph of the COBOL 5 source program.

The COBOL 5 source program containing the file description for this data file is also a NOS permanent file; the file name for the source program file is SOURCE.

First, log in to NOS. Then attach the COBOL 5 data file to be migrated and the file containing the COBOL 5 source program. Then log in to NOS/VE and enter the following commands:

```
/get_file to=source
/create_file file=master
/set_file_attributes file=master ..
.. /block_type=system_specified ..
.. /record_type=variable
/open_file_migration_aid
FA/collect_file_description input=source ..
                            assigned_name=master ..
                            output=description
FA/migrate_file input_file_description=description ..
                           list=changes
FA/close_environment
```

#### GET_FILE

The command:

```
/get_file  to=source
```

in this example copies file SOURCE from NOS to NOS/VE. In this example, file SOURCE contains the COBOL 5 source program.

### CREATE_FILE

The command:

```
/create_file  file=master
```

in this example creates an empty file on NOS/VE by the name MASTER. This file will later contain the migrated version of the COBOL 5 data file.

### SET_FILE_ATTRIBUTES

The command:

```
/set_file_attributes  file=master ..
.. /block_type=system_specified ..
.. /record_type=variable
```

ensures that the file holding the migrated version of the COBOL 5 data file has the correct block type and record type.

### OPEN_FILE_MIGRATION_AID

The command:

```
/open_file_migration_aid
```

in this example initiates the COBOL FMA utility. This command opens a migration environment; that is, issuing this command provides you with access to a number of other commands that you will use to migrate the file.

### COLLECT_FILE_DESCRIPTION

The command:

```
FA/collect_file_description input=source ..
                            assigned_name=master ..
                            output=description
```

in this example extracts the file description with the ASSIGNED_NAME of MASTER from the COBOL 5 source program. This gets the file description from the COBOL 5 program on file SOURCE and places it on file DESCRIPTION.

### MIGRATE_FILE

The command:

```
FA/migrate_file input_file_description=description ..
                list=changes
```

in this example causes the COBOL 5 data file MASTER to be migrated from NOS to NOS/VE. The migrated file is placed on NOS/VE file MASTER. This command also produces file CHANGES, which contains a listing; this listing summarizes any changes that you must make to the file description in the COBOL source program in order to use it on NOS/VE.

Notice that this command does not specify the file name of the NOS file being migrated. The default is used instead; the default is the name assigned by the ASSIGN clause in the file description.

### CLOSE_ENVIRONMENT

The command:

```
FA/close_environment
```

in this example closes the COBOL FMA environment.

### Example of Using a Record Procedure

This example demonstrates how to migrate a COBOL 5 data file from NOS to NOS/VE. In this example, the file description for the data file to be migrated contains an ambiguity: the file description redefines items. Therefore, a record procedure is required.

The data file to be migrated is a NOS permanent file; the file name for the data file is OM12345. However, the name assigned in the ASSIGN clause in the file description paragraph of the COBOL 5 source program is MASTER. Also, the record description does not reflect the true CRM record type on the file to be migrated.

The COBOL 5 source program containing the file description for this data file is also a NOS permanent file; the file name for the source program file is SOURCE.

The following is the file description in the COBOL 5 source program. Notice the ASSIGN clause. Also notice the REDEFINES clause.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  FMA-FILE-DESCRIPTION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PAYROLL-MASTER-170 ASSIGN TO MASTER.   <----------------------
DATA DIVISION.
FILE SECTION.
.
.
.


FD PAYROLL-MASTER-170; LABEL RECORD IS OMITTED.
01 EMPLOYEE-RECORD.
    02 EMPLOYEE-IDENTIFICATION.
        03 SOC-SEC-NUMBER PIC X(9).
        03 EMPLOYEE-NUMBER PIC X(5).
    02 EMPLOYEE-NAME.
        03 FIRST-NAME PIC X(20).
        03 MIDDLE-INITIALS PIC XX.
        03 FAMILY-NAME PIC X(30).
    02 GRADE PIC 99.
    02 DIVISION-NUMBER PIC 9(4).
    02 EXEMPT-DATA.
        03 BONUS-PLAN PIC 99.
        03 MONTHLY-SALARY PIC 9(5)V99 COMP-1.
        03 ACCUM-PAYMENTS PIC 9(6)V99 COMP-1.
    02 NON-EXEMPT-DATA REDEFINES EXEMPT-DATA.   <-------------------------
        03 HRLY-RATE PIC 99V99.
        03 UNION-ID PIC XXX.
        03 STD-HRS-PER-PERIOD PIC 99V9 COMP-4.
        03 ACCUM-PAYMENTS PIC 9(5)V99 COMP-1.
```

Log in to NOS/VE and enter the following commands:

```
/get_file  to=source  data_conversion=D64
/create_file  file=master
/set_file_attributes  file=master ..
.. /block_type=system_specified ..
.. /record_type=variable
/collect_text  output=process
ct? employee-record
ct? if grade is greater than 9
ct?     then exempt-data
ct?     else non-exempt-data
ct? endif
ct? **
/open_file_migration_aid
FA/open_170_state
FA/    execute_command  command='FILE,MASTER,BT=I,RT=W.'
FA/close_environment
FA/collect_file_description input=source ..
                            assigned_name=master ..
                            output=description
FA/migrate_file input=om12345 ..
                input_file_description=description ..
                record_procedure=process
FA/close_environment
```

### GET_FILE

The command:

```
/get_file  to=source  data_conversion=d64
```

in this example copies file SOURCE from NOS to NOS/VE. In this example, file SOURCE contains the COBOL 5 source program.

### CREATE_FILE

**The command:**

```
/create_file  file=master
```

in this example creates an empty file on NOS/VE by the name MASTER. This file will later contain the migrated version of the COBOL 5 data file.

### SET_FILE_ATTRIBUTES

**The command:**

```
/set_file_attributes  file=master ..
.. /block_type=system_specified ..
.. /record_type=variable
```

ensures that the file holding the migrated version of the COBOL 5 data file has the correct block type and record type.

## COLLECT_TEXT

The command:

```
/collect_text  output=process
```

in this example creates a file named PROCESS and places text on that file. After entering the COLLECT_TEXT command, you are prompted for each line of text. Enter one line at a time, pressing the carriage return after each line. When you are finished entering text, enter two asterisks in response to the COLLECT_TEXT prompt and press the carriage return. (The asterisks are not placed on the file.)

The data entered into file PROCESS in this example is a record procedure. The following screens explain the commands in the record procedure.

The record procedure is required because the file description redefines items; as you can see, NON-EXEMPT-DATA redefines EXEMPT-DATA:

```
01 EMPLOYEE-RECORD
.
.
.
    02 GRADE PIC 9(4).
     .
     .
     .
    02 EXEMPT-DATA.
     .
     .
     .
    02 NON-EXEMPT-DATA REDEFINES EXEMPT-DATA.   <-------------------------
     .
     .
```

The first statement in the record procedure is:

```
employee-record
```

This is a conversion reference that migrates EMPLOYEE-RECORD.

The IF construct in the record procedure chooses either EXEMPT-DATA or NON-EXEMPT-DATA, depending on the value of GRADE.

```
if grade is greater than 9
     then exempt-data
     else non-exempt-data
endif
```

If GRADE is greater than 9, then EXEMPT-DATA is used; otherwise, NON-EXEMPT-DATA is used.

## OPEN_FILE_MIGRATION_AID

The command:

```
/open_file_migration_aid
```

in this example initiates the COBOL FMA utility. This command opens a migration environment; that is, issuing this command provides you with access to a number of other commands that you will use to migrate the file.

### OPEN_170_STATE

The command:

```
FA/open_170_state
```

in this example creates a connection to the NOS operating system enabling you to execute NOS commands.

The reason that a NOS command must be executed in this example is because the record description does not reflect the true CRM record type on the file to be migrated.

### EXECUTE_COMMAND

The command:

```
FA/execute_command  command='FILE,MASTER,RT=W.'
```

in this example executes the NOS command:

```
FILE,MASTER,RT=W.
```

This NOS command changes the record description of the file to reflect the true CYBER Record Manager record type on file MASTER.

### CLOSE_ENVIRONMENT

The command:

```
FA/close_environment
```

in this example closes the 170 state environment. That is, the connection that enables you to execute NOS commands is broken.

After executing this command, you can no longer execute the EXECUTE_COMMAND command unless you reopen 170 state. You remain in the COBOL FMA environment.

### COLLECT_FILE_DESCRIPTION

The command:

```
FA/collect_file_description input=source ..
                            assigned_name=master ..
                            output=description
```

in this example extracts the file description with the ASSIGNED_NAME of MASTER from the COBOL 5 source program. This gets the file description from the COBOL 5 program on file SOURCE and places it on file DESCRIPTION.

### MIGRATE_FILE

The command:

```
FA/migrate_file  input=om12345 ..
    input_file_description=description ..
    record_procedure=process
```

in this example causes the COBOL 5 data file MASTER to be migrated from NOS to NOS/VE. The migrated file is placed on NOS/VE file MASTER. This command also specifies the name of the file containing the record procedure to be used in the migration process. This is file PROCESS, which was created previously using the COLLECT_TEXT command.

Notice that this command does not specify the file name of the NOS file being migrated. Instead, the default is used for the INPUT parameter; the default is the file name assigned by the ASSIGN clause in the file description and is the same as the file name.

### CLOSE_ENVIRONMENT

The command:

```
FA/close_environment
```

in this example closes the COBOL FMA environment.

After executing this command, you can no longer execute COBOL FMA commands unless you reopen FMA. You are returned to the NOS/VE command environment.

### Reverse Migration Example

The following example illustrates the migration of a NOS/VE file to NOS. The example assumes the following:

- The file is described in a COBOL 5-compatible source program (which can be compiled with BASE_LANGUAGE=COBOL5 on the COBOL command). The program is on a NOS/VE permanent file named $USER.COBOL_SOURCE.
- The file description contains no ambiguities; that is, it has a single 01 descriptor and no REDEFINES clause. Therefore, no record procedure is needed.
- The FD entry for the file is:

```
SELECT MYFILE ASSIGN TO "VEFILE".
```

The following commands migrate the NOS/VE file to NOS:

```
/open_file_migration_aid                    "CALL FMA"

FA/collect_file_description ..
    input=$user.cob_source ..               "File containing COBOL 5 source"
    assigned_name=vefile ..                 "File name in ASSIGN clause. (Identifies ..
                                               the description to be copied)" ..
    output=myfile_descr                     "File to receive file description."

FA/migrate_file ..
    input_file_description=(myfile_descr, ..  "Read file description from MYFILE_DESCR," ..
      c180_cobol_cc) ..                     "  and migrate NOS/VE-to-NOS" ..
    input=$user.vefile ..                   "File to be migrated" ..
    output=nosfile ..                       "Migrated file on NOS"

FA/close_environment                        "End FMA"
```

Following are detailed descriptions of the preceding commands.

```
/open_file_migration_aid
```

This command opens the migration environment. After you enter this command, you can enter FMA commands to migrate the file.

```
FA/collect_file_description ..
    input=$user.cob_source ..
    assigned_name=vefile ..
    output=myfile_descr
```

This command extracts the description of the NOS/VE file from the COBOL source program. The source program is on file COB_SOURCE. FMA selects the file description of the file having the assigned name VEFILE (specified in the SELECT clause of the FILE-CONTROL paragraph) and copies that description to file MYFILE_DESCR.

```
FA/migrate_file ..
    input_file_description=(myfile_descr, c180_cobol_cc) ..
    input=$user.vefile ..
    output=nosfile
```

This command migrates the file. FMA reads the file description from file MYFILE_DESCR, and migrates the NOS/VE file VEFILE to the NOS file NOSFILE.

```
FA/close_environment
```

This command ends the FMA task.

## FMA Performance Considerations

The performance of FMA is influenced by the options and statements you specify in an FMA task (although the effect on performance is generally minor).

The following are guidelines for obtaining maximum performance from FMA.

- Migrating a FORTRAN file is generally slower than doing an extended access read on the file. This is because for file migration, a record must be written for each record read.
- Several EXECUTE_MIGRATION_TASK command parameters are identical to parameters for the SCL EXECUTE_TASK command. Performance considerations are the same for both sets of parameters.
- Using record procedures when migrating a COBOL file decreases FMA performance. Therefore, you should use record procedures only where necessary; that is, when file descriptions contain multiple 01 specifications or REDEFINES clauses.
- Variable length records require more execution time to migrate than fixed length records.
- Specifying multiple NOS commands on a single EXECUTE_COMMAND command is significantly faster than specifying a separate EXECUTE_COMMAND for each NOS command.
- The selection of job statement parameters for the PARTNER_JOB_CARD parameter on the OPEN_FILE_MIGRATION_AID command can affect performance. Refer to Volume 3 of the NOS reference manual for descriptions of the Job card parameters.
- You should use conversion references in record procedures to reference the highest level (lowest level numbers) at which the file description is ambiguous. To do otherwise decreases performance.
- In record procedures, you should use the $INTEGER function only to test fields that contain binary data and are whole word fields beginning on a word boundary.

# Migrating Tape Files

You can use FMA or FMU to migrate certain NOS tape files to NOS/VE.

## Introduction to Tape File Migration

On dual state systems there are two ways to migrate tape files. The first way is to migrate the file through an interstate connection. You can use FMU, FORTRAN FMA, and COBOL FMA in this way. Migrating a tape file through an interstate connection is identical to migrating a disk file, except that you must request the tape (by specifying a REQUEST statement) before migrating the tape file. You can execute a REQUEST statement from the NOS/VE side of a dual state system through the EXECUTE_INTERSTATE_COMMAND (EXEIC) command. For example:

```
/create_interstate_connection
/execute_interstate_command  command='REQUEST,TAPE3,PE,US,VSN=VOL001.'
```

The CREATE_INTERSTATE_CONNECTION command opens the interstate connection. The EXECUTE_INTERSTATE_COMMAND command specifies a REQUEST statement to be executed on the NOS side of the dual state system. File TAPE3 can then be migrated by using FMA or FMU as described earlier in this manual.

The second way of migrating a NOS tape file is to read the file from a NOS/VE tape drive. You can read the tape file directly from NOS/VE, without going through an interstate connection. NOS/VE can handle the NOS tape file on a NOS/VE tape drive by using the following four commands:

CREATE_170_REQUEST

CHANGE_170_REQUEST

DISPLAY_TAPE_LABEL_ATTRIBUTES

DETACH_FILE

The CREATE_170_REQUEST command creates a temporary NOS/VE file and associates it with a NOS tape file on a NOS/VE tape drive. The temporary file describes the NOS tape file in terms that NOS/VE can understand. The command does not cause NOS tape file access or operator communication for tape mounting. You reference the temporary NOS/VE file in a later SCL command to actually request and read the NOS tape file.

The CHANGE_170_REQUEST command allows you to change a NOS tape file description in a temporary NOS/VE file formed in a preceding CREATE_170_REQUEST command.

You use the DISPLAY_TAPE_LABEL_ATTRIBUTES command for informational purposes. DISPLAY_TAPE_LABEL_ATTRIBUTES works only for ANSI labeled tapes. It can display information about a NOS tape file from the associated temporary NOS/VE file.

You use the DETACH_FILE command to delete/detach the temporary file associated with the NOS tape file. This command also releases the physical drive.

You can find quick reference format information about any of these commands by using the DISPLAY_COMMAND_INFORMATION command. For example, to learn about the CREATE_170_REQUEST command, enter:

```
/display_command_information  command=create_170_request
```

In addition, the SCL System Interface Usage manual for NOS/VE release 1.2.1, PSR level 670 or later, describes the DISPLAY_TAPE_LABEL_ATTRIBUTES and DETACH_FILE commands in detail.

The following material includes:

A detailed description of the CREATE_170_REQUEST and CHANGE_170_REQUEST commands with examples

A brief description of the DISPLAY_TAPE_LABEL_ATTRIBUTES and DETACH_FILE commands, with examples of their use with NOS tape files

In addition, there is an example of using the tape migration commands in each of the following migration situations:

Migrating a character data tape file

Migrating a tape file with binary data

Migrating several tape files on the same multifile set

## CREATE_170_REQUEST

The CREATE_170_REQUEST command creates a NOS/VE temporary file to be associated with a NOS tape file. Future references to the NOS tape file are through the NOS/VE temporary file.

### CREATE_170_REQUEST Format

The CREATE_170_REQUEST command has the following format:

```
CREATE_170_REQUEST or
CRE1R
    FILE=file
    EXTERNAL_VSN=list of string
    RECORDED_VSN=list of string
    FILE_SET_POSITION=keyword
    FILE_IDENTIFIER=string
    FILE_SEQUENCE_NUMBER=integer
    GENERATION_NUMBER=integer
    INTERNAL_CODE=keyword
    CHARACTER_CONVERSION=boolean
    BLOCK_TYPE=keyword
    RECORD_TYPE=keyword
    MAXIMUM_BLOCK_LENGTH=integer
    MAXIMUM_RECORD_LENGTH=integer
    LABEL_TYPE=keyword
    TAPE_FORMAT=keyword
    STATUS=status variable
```

The FILE (F) parameter is required. It specifies the name of a NOS/VE temporary file to be associated with a NOS tape file.

The EXTERNAL_VSN (EVSN or VSN) parameter is optional. It gives the external identification of the tape volume(s) containing the NOS tape file. Each parameter value is a string 1 to 6 characters long.

If you specify more than one external volume serial number (VSN), the volumes are requested in the order specified in the parameter list. If you omit the EXTERNAL_VSN parameter, the system uses the RECORDED_VSN parameter in its stead.

You must specify either the EXTERNAL_VSN parameter or the RECORDED_VSN parameter. Otherwise, a fatal error results.

The RECORDED_VSN (RVSN) parameter is optional. It is for ANSI labeled tapes only. If you enter the RECORDED_VSN parameter for an unlabeled tape, the parameter is ignored, unless there is no matching EXTERNAL_VSN parameter. In this case, the RECORDED_VSN parameter takes on the functions of the EXTERNAL_VSN parameter.

The RECORDED_VSN parameter gives the VSN recorded internally on the ANSI VOL1 label on the tape volume(s) holding the NOS tape file. Each parameter value is a string 1 to 6 characters long. File processing uses the RECORDED_VSN parameter to locate and verify the correct volume.

If you specify more than one recorded VSN, the volumes are located and verified in the order specified in the parameter list. If you omit the RECORDED_VSN parameter for an ANSI labeled tape, the system uses the EXTERNAL_VSN parameter to verify the VSN recorded internally on the ANSI VOL1 tape label.

If you specify both the EXTERNAL_VSN and RECORDED_VSN parameters, they are matched; the first external VSN with the first recorded VSN, the second external VSN with the second recorded VSN, and so on. For each such pair, NOS/VE uses the external VSN parameter to direct the system operator to mount the tape with that external VSN. For an ANSI labeled tape, NOS/VE uses the recorded VSN value to verify the VSN recorded internally on the ANSI VOL1 label on that tape.

If there is an EXTERNAL_VSN parameter with no matching RECORDED_VSN parameter, NOS/VE uses the EXTERNAL_VSN parameter to direct the system operator. For an ANSI labeled tape, NOS/VE also uses the EXTERNAL_VSN parameter to verify the VSN recorded internally on the ANSI VOL1 tape label.

If there is a RECORDED_VSN parameter with no matching EXTERNAL_VSN parameter, NOS/VE uses the RECORDED_VSN parameter to direct the system operator. For an ANSI labeled tape, NOS/VE also uses the RECORDED_VSN parameter to verify the VSN recorded internally on the ANSI VOL1 tape label.

The FILE_SET_POSITION (FSP) parameter is optional. It specifies the position of the NOS tape file to be read.

The FILE_SET_POSITION parameter is not needed for unlabeled tapes because NOS/VE assumes that you wish to read the first file on an unlabeled tape. That is, the value of the FILE_SET_POSITION parameter for a NOS tape file on an unlabeled tape is BEGINNING_OF_SET, regardless of what you enter.

Only labeled tapes can use all the values of the FILE_SET_POSITION parameter. If you omit the parameter for a labeled tape, the NEXT_FILE position is assumed.

The parameter can have any of the following values:

BEGINNING_OF_SET (BOS)

This value specifies that the first tape file on the file set is to be read.

CURRENT_FILE (CF)

This value specifies that the current tape file is to be read. That is, the last tape file accessed will be accessed again. If the tape is positioned at the beginning of the first volume, the first tape file will be read.

FILE_IDENTIFIER_POSITION (FIP)

This value specifies that the tape file identified by the FILE_IDENTIFIER and GENERATION_NUMBER parameters is to be read.

FILE_SEQUENCE_POSITION (FSP)

This value specifies that the tape file identified by the FILE_SEQUENCE_NUMBER parameter is to be read.

NEXT_FILE (NF)

This value specifies that the tape file following the last accessed tape file will be read. If the tape is positioned at the beginning of the first volume, the first tape file will be read.

The FILE_IDENTIFIER (FI) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value is a string of 1 to 17 characters that specifies a file identifier. Each tape file on a multifile set has a unique file identifier. If you specify the FILE_IDENTIFIER_POSITION value for the FILE_SET_POSITION parameter, the FILE_IDENTIFIER parameter is required; otherwise, its value is ignored.

The FILE_SEQUENCE_NUMBER (FSN) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value is an unsigned integer in the range 1 through 9999 that specifies the numeric position of a tape file on a multifile set. If you specify the FILE_SEQUENCE_POSITION value for the FILE_SET_POSITION parameter, the FILE_SEQUENCE_NUMBER parameter is required; otherwise, its value is ignored.

The GENERATION_NUMBER (GN) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value, an unsigned integer in the range 1 through 9999, identifies a specific revision of the tape file named by the FILE_IDENTIFIER parameter. If the FILE_SET_POSITION parameter has the FILE_IDENTIFIER_POSITION value, and the GENERATION_NUMBER parameter is omitted, then the GENERATION_NUMBER parameter value is set to one.

The INTERNAL_CODE (IC) parameter is optional. It specifies the character set of the data on the tape volume. If you omit the parameter, its value is set to D64. The INTERNAL_CODE parameter can have the following values:

AS6 6/12-bit ASCII

AS8 8/12-bit ASCII

D63 63-character display code

D64 64-character display code

The CHARACTER_CONVERSION (CC) parameter is optional. Its boolean value specifies whether or not file data is to be converted to or from the character set specified by the INTERNAL_CODE parameter. If you omit the CHARACTER_CONVERSION parameter, FALSE is assumed to be its value.

Of the tape file migration methods, only FMA and FMU automatically do character conversion in addition to any conversion specified by the CHARACTER_CONVERSION parameter value.

To obtain a properly migrated tape file, you usually want to set the CHARACTER_CONVERSION parameter to:

FALSE if you use FMA or FMU to migrate. Otherwise you convert your tape file data twice.

TRUE if you use any other tape file migration method. Otherwise, you do not convert your tape file data at all.

The BLOCK_TYPE (BT) parameter is optional. It specifies the block type of the NOS input tape file. If you omit this parameter, its value is set to INTERNAL. Its value can be either of the following:

| | |
|---|---|
| INTERNAL (I) | Internal blocking |
| CHARACTER_COUNT (CC) | Character count blocking |

The RECORD_TYPE (RT) parameter is optional. Its value specifies the record type of the NOS tape file. If you omit this parameter, its value is set to CONTROL_WORD. Its value can be any of the following:

| | |
|---|---|
| CONTROL_WORD (CW or W) | Control word |
| FIXED_LENGTH (FL or F) | Fixed length |
| SYSTEM_RECORD (SR or S) | System record |
| ZERO_BYTE (ZB or Z) | Zero-byte |

The MAXIMUM_BLOCK_LENGTH (MAXBL or MBL) parameter is optional. Its value is an unsigned integer that specifies the maximum length in 6-bit bytes of a block in the NOS tape file. The system maximum for this parameter is 2,147,483,615. If you omit this parameter, its value is set to 5120.

The MAXIMUM_RECORD_LENGTH (MAXRL or MRL) parameter is optional. Its value is an unsigned integer that specifies the maximum length in 6-bit bytes of a record in the NOS tape file. The system maximum for this parameter is 4,398,046,511,103. If you omit this parameter, its value is set to 5120.

The LABEL_TYPE (LT) parameter is optional. Its value specifies whether the tape is labeled. If you omit this parameter, its value is set to STANDARD. This parameter can have the following values:

| | |
|---|---|
| LABELLED (L) | Same as STANDARD. |
| STANDARD (S) | Tape has standard labels. |
| UNLABELLED (U) | Tape is not labeled. |

The TAPE_FORMAT (TF) parameter is optional. It specifies the tape format of the NOS tape file. If you omit this parameter, is value is set to NOS_INTERNAL. The possible values for this parameter are:

| | |
|---|---|
| NOS_INTERNAL (NI or I) | Internal, NOS default tape format |
| STRANGER (S) | Stranger |
| LONG_STRANGER (LS or L) | Long block stranger |

The STATUS variable is optional. It is the standard SCL status variable. Refer to chapter 2 for more information.

### Restrictions on the NOS Tape Files

There are restrictions and considerations in using the CREATE_170_REQUEST command. They are divided as follows:

Tape Characteristics

Labeled and Unlabeled Tape Differences

Tape File Configurations

FMA and FMU Considerations

Positioning the NOS Tape File

Output File Characteristics

### Tape Characteristics

The CREATE_170_REQUEST command supports standard labeled and unlabeled binary mode tapes. The tapes are nine track and can have one of the following densities:

800 characters per inch (cpi)

1600 cpi

6250 cpi

You do not need to specify the density of the tape containing the NOS tape file. The system can determine this information.

There are four possible combinations of block type and record type:

| Block Type | Record Type |
|---|---|
| character-count | fixed-length |
| character-count | system-record |
| character-count | zero-byte |
| internal | control-word |

### Labeled and Unlabeled Tape Differences

If you attempt to process a labeled tape as an unlabeled tape, results are unpredictable.

The following restrictions apply to unlabeled tapes:

Mixed tape formats are not supported.

Multifile sets are not supported.

If the tape has records with the control word record type (RECORD_TYPE=CONTROL_WORD) and the tape format is long block stranger or stranger, then all the files on the tape must have records with the same record type.

### Tape File Configurations

The CREATE_170_REQUEST command can create a NOS/VE temporary file referencing any of the following kinds of NOS tape files:

- A single file on a single volume
- A single file spanning several volumes
- A single file within a multifile set on a single volume
- A single file within a multifile set spanning several volumes

### FMA and FMU Considerations

You must enter a CREATE_170_REQUEST command before using FMA or FMU to migrate the NOS tape file.

If you are using FMA, you specify the name of the NOS/VE temporary file in the MIGRATION_FILES or EXTENDED_ACCESS_FILES parameter of the EXECUTE_MIGRATION_TASK command. When the FORTRAN program executing under FMA opens the temporary file, the NOS tape file is requested.

If you are using FMU, you specify the name of the temporary file on the SET_INPUT_ATTRIBUTES directive. You also specify MACHINE_FORMAT=C170 in the SET_INPUT_ATTRIBUTES directive. When you initiate FMU, the tape request is issued and the NOS tape file is read.

FMA and FMU automatically do character conversion in addition to any conversion specified by the CHARACTER_CONVERSION parameter value. If you use FMA or FMU to migrate, you usually want to set the CHARACTER_CONVERSION parameter to FALSE. Otherwise you convert your tape file data twice.

For standard labeled tapes, FMA and FMU can do the following:

Read a specific file in a file set. (You specify the file sequence number or the file identifier in the CREATE_170_REQUEST command.)

Read all sections of a file in single-file/single-volume, single-file/multivolume, multifile/single-volume, and multifile/multivolume configurations.

### Positioning the NOS Tape File

If a NOS tape file is to be positioned to a partition boundary, all data on the file must have the same tape format. The tape formats are listed in the CREATE_170_REQUEST format description earlier in this chapter.

You can position to a partition boundary by using the STARTING_FILE_POSITION parameter on the SET_INPUT_ATTRIBUTES FMU directive or by using the FORTRAN EOF function when using FMA. Refer to the description of FMU in the SCL Advanced File Management Usage manual for a description of the STARTING_FILE_POSITION parameter. The EOF function is described in the FORTRAN Language Definition Usage manual.

After the NOS file on the labeled tape is located, you can position to a partition boundary. Unlabeled tapes cannot be positioned by file sequence number. Unlabeled tapes can be positioned only by partitions, using the STARTING_FILE_POSITION parameter with FMU or the FORTRAN EOF function with FMA.

### Output File Characteristics

Not previously set file attributes of the NOS/VE file receiving the migrated NOS tape file get their values from the temporary NOS/VE file associated with the NOS tape file. The values of two of these file attributes can cause fatal errors when the new NOS/VE file is referenced. To ensure that these file attributes have correct values, enter the following command before the command that migrates the NOS tape file:

```
SET_FILE_ATTRIBUTES  FILE=file ..
    BLOCK_TYPE=keyword ..
    RECORD_TYPE=keyword
```

Here FILE or F specifies the NOS/VE file receiving the migrated NOS tape file.

The BLOCK_TYPE or BT parameter can have one of the following values:

| | |
|---|---|
| SYSTEM_SPECIFIED or SS | The file is logically divided into a number of fixed-size blocks whose length is determined by NOS/VE. |
| USER_SPECIFIED or US | The file is logically divided into a number of blocks whose length can vary between a user-defined minimum and maximum length. |

The RECORD_TYPE or RT parameter can have one of the following values:

| | |
|---|---|
| VARIABLE or V | CDC variable |
| FIXED or F | ANSI fixed length |
| UNDEFINED or U | Undefined |

For more information about which values to use for block type and record type, see the CYBIL File Interface Usage manual or the appropriate programming language usage manual.

### CREATE_170_REQUEST Examples

There are five examples of the CREATE_170_REQUEST (CRE1R):

CRE1R for single tape file on single volume

CRE1R for single FORTRAN tape file on several volumes

CRE1R for FORTRAN tape file on multifile set on several volumes

CRE1R for generation 1 tape file on multifile set on one volume,

CRE1R for generation 3 tape file on multifile set on one volume

### CRE1R, One File, One Volume

The following command associates the NOS/VE temporary file ONE_FILE with a single NOS tape file on a single volume with external VSN of 3193:

```
/create_170_request  file=one_file ..
.. /external_vsn='3193' ..
.. /file_set_position=next_file ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /label_type=standard ..
.. /tape_format=nos_internal
```

NOS/VE directs the system operator to mount the volume with external VSN 3193. Because the tape is an ANSI labeled tape, the system verifies that the VSN recorded internally on the ANSI VOL1 label is 3193.

The example tells that the NOS tape file:

Is the tape file following the last accessed tape file (FILE_SET_POSITION=NEXT_FILE). If the tape is positioned at the beginning of the first volume, the first tape file is read.

Has character data in the 64-character display code character set, the default (INTERNAL_CODE=D64)

Has the default character conversion (CHARACTER_CONVERSION=FALSE)

Has the default blocking type, internal blocking (BLOCK_TYPE=INTERNAL)

Has the default record type, control word (RECORD_TYPE=CONTROL_WORD)

Has the default maximum block length of 5120 (MAXIMUM_BLOCK_LENGTH=5120)

Has the default maximum record length of 5120 (MAXIMUM_RECORD_LENGTH=5120)

Is labeled, the default (LABEL_TYPE=STANDARD)

Has the default tape format, internal (TAPE_FORMAT=NOS_INTERNAL)

By omitting the parameters whose values were defaults, you can shorten the command to:

```
/create_170_request  file=one_file ..
.. /external_vsn='3193'
```

### CRE1R, One FORTRAN File, Several Volumes

This example associates the temporary file SOME_VOLUMES with a typical FORTRAN formatted sequential, namelist, or list directed file (BLOCK_TYPE=CHARACTER_COUNT, RECORD_TYPE=ZERO_BYTE). The file is a single NOS tape file spanning two volumes:

```
/create_170_request  file=some_volumes ..
.. /external_vsn=('3193'  '1019') ..
.. /recorded_vsn=('iv01'  'iv02') ..
.. /file_set_position=current_file ..
.. /internal_code=d63 ..
.. /character_conversion=true ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=4000 ..
.. /maximum_record_length=200
```

NOS/VE directs the system operator to mount the volumes with external VSNs 3193 and 1019. Because the tapes are ANSI labeled tapes, the system verifies that the VSNs recorded internally on the ANSI VOL1 labels are iv01 and iv02.

The example tells that the NOS tape file:

Is the current tape file (FILE_SET_POSITION=CURRENT_FILE); that is, the last tape file accessed. If the tape is positioned at the beginning of the first volume, the first tape file is read.

Has character data in the 63-character display code character set (INTERNAL_CODE=D63)

Is to be converted from the D63 character set (CHARACTER_CONVERSION=TRUE)

Has character count blocking (BLOCK_TYPE=CHARACTER_COUNT)

Has the zero-byte record type (RECORD_TYPE=ZERO_BYTE)

Has maximum block length of 4000 (MAXIMUM_BLOCK_LENGTH=4000)

Has maximum record length of 200 (MAXIMUM_RECORD_LENGTH=200)

Is labeled, the default

Has the default tape format, internal

### CRE1R, FORTRAN File on Multifile Set, Several Volumes

This next command associates temporary file FILE_5 with a typical FORTRAN unformatted sequential file (BLOCK_TYPE=INTERNAL, RECORD_TYPE=CONTROL_WORD). The file is the fifth tape file in a multifile set that spans two volumes:

```
/create_170_request  file=file_5 ..
.. /external_vsn=('3907'  '1007') ..
.. /recorded_vsn=('7110'  '5811') ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=5
```

NOS/VE directs the system operator to mount the volumes with external VSNs 3907 and 1007. Because the tapes are ANSI labeled tapes, the system verifies that the VSNs recorded internally on the ANSI VOL1 labels are 7110 and 5811.

The example tells that the NOS tape file:

Is to be found by its position on the tape (FILE_SET_POSTION=FILE_SEQUENCE_POSITION)

Is the fifth file on the multifile set (FILE_SEQUENCE_NUMBER=5)

Has character data in the 64-character display code character set, the default

Has the default character conversion, blocking, record type, maximum block length, maximum record length, and tape format

Is labeled, the default

### CRE1R, Generation 1 File on Multifile Set, One Volume

The following command associates temporary file AB_CD_GOLDFISH with a NOS tape file with identifier LITTLE_CARP:

```
/create_170_request  file=ab_cd_goldfish ..
.. /external_vsn='h20' ..
.. /recorded_vsn='water' ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='little_carp'
```

NOS/VE directs the system operator to mount the volume with external VSN H20. Because the tape is an ANSI labeled tape, the system verifies that the VSN recorded internally on the ANSI VOL1 label is WATER.

The example tells that the NOS tape file:

Is to be found from its file identifier and generation number (FILE_SET_POSITION= FILE_IDENTIFIER_POSITION)

Is identified within the multifile set as LITTLE_CARP (FILE_IDENTIFIER=LITTLE_CARP)

Is the first generation of LITTLE_CARP, because the default value of GENERATION_NUMBER is 1

Has character data in the 64-character display code character set, the default

Has the default character conversion, blocking, record type, maximum block length, maximum record length, and tape format

Is labeled, the default

### CRE1R, Generation 3 File on Multifile Set, One Volume

The following command associates temporary file LM_NO_GOLDFISH with a NOS tape file that is the third cycle of a file identified as BIG_GUPPY:

```
/create_170_request  file=lm_no_goldfish ..
.. /external_vsn='agua' ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='big_guppy' ..
.. /generation_number=3
```

NOS/VE directs the system operator to mount the volume with external VSN AGUA. Because the tape is an ANSI labeled tape, the system verifies that the VSN recorded internally on the ANSI VOL1 label is AGUA.

The example tells that the NOS tape file:

Is to be found from its file identifier and generation number (FILE_SET_POSTION= FILE_IDENTIFIER_POSITION)

Is identified within the multifile set as BIG_GUPPY (FILE_IDENTIFIER=BIG_GUPPY)

Is the third generation of BIG_GUPPY (GENERATION_NUMBER=3)

Has character data in the 64-character display code character set, the default

Has the default character conversion, blocking, record type, maximum block length, maximum record length, and tape format

Is labeled, the default

## CHANGE_170_REQUEST

The CHANGE_170_REQUEST command changes the description of a NOS tape file. The command operates on the NOS/VE temporary file associated with the NOS tape file.

This command is handy when you wish to use the same temporary NOS/VE file to reference several tape files from the same multifile set on tape. You simply use the CHANGE_170_REQUEST command to change the tape file description.

In general, when you omit a parameter from the CHANGE_170_REQUEST command, the value of that parameter is not changed. Thus, usually, you need to include only those parameters whose settings change for the new NOS tape file.

The two exceptions to this rule are the FILE_SET_POSITION and the GENERATION_NUMBER parameters. When you omit these parameters, their values become the default values of the parameters. For the FILE_SET_POSITION parameter, the default is BEGINNING_OF_SET for an unlabeled tape and NEXT_FILE for a labeled tape. The default value for the GENERATION_NUMBER parameter is 1.

As with the CREATE_170_REQUEST command, not previously set file attributes of the NOS/VE file receiving the migrated NOS tape file get their values from the temporary NOS/VE file associated with the NOS tape file. The values of two of these file attributes can cause fatal errors when the new NOS/VE file is referenced. To ensure that these file attributes have correct values, enter the following command before the command that migrates the NOS tape file:

```
SET_FILE_ATTRIBUTES  FILE=file ..
    BLOCK_TYPE=keyword ..
    RECORD_TYPE=keyword
```

Here FILE or F specifies the NOS/VE file receiving the migrated NOS tape file.

For more information about which values to use for BLOCK_TYPE and RECORD_TYPE, see the CYBIL File Interface Usage manual or the appropriate programming language usage manual.

### CHANGE_170_REQUEST Format

The CHANGE_170_REQUEST command has the following format:

```
CHANGE_170_REQUEST or
CHA1R
    FILE=file
    FILE_SET_POSITION=keyword
    FILE_IDENTIFIER=string
    FILE_SEQUENCE_NUMBER=integer
    GENERATION_NUMBER=integer
    INTERNAL_CODE=keyword
    CHARACTER_CONVERSION=boolean
    BLOCK_TYPE=keyword
    RECORD_TYPE=keyword
    MAXIMUM_BLOCK_LENGTH=integer
    MAXIMUM_RECORD_LENGTH=integer
    TAPE_FORMAT=keyword
    STATUS=status variable
```

The FILE (F) parameter is required. It specifies the name of a NOS/VE temporary file associated with a NOS tape file by a previous CREATE_170_REQUEST command.

The FILE_SET_POSITION (FSP) parameter is optional. It specifies the position of the NOS tape file to be read.

The FILE_SET_POSITION parameter is not needed for unlabeled tapes because NOS/VE assumes that you wish to read the first file on an unlabeled tape. That is, the value of the FILE_SET_POSITION parameter for a NOS tape file on an unlabeled tape is BEGINNING_OF_SET, regardless of what you enter.

Only labeled tapes can use all the values of the FILE_SET_POSITION parameter. If you omit the parameter for a labeled tape, the NEXT_FILE position is assumed.

The parameter can have any of the following values:

BEGINNING_OF_SET (BOS)

This value specifies that the first tape file on the file set is to be read.

CURRENT_FILE (CF)

This value specifies that the current tape file is to be read. That is, the last tape file accessed will be accessed again. If the tape is positioned at the beginning of the first volume, the first tape file will be read.

FILE_IDENTIFIER_POSITION (FIP)

This value specifies that the tape file identified by the FILE_IDENTIFIER and GENERATION_NUMBER parameters is to be read.

FILE_SEQUENCE_POSITION (FSP)

This value specifies that the tape file identified by the FILE_SEQUENCE_NUMBER parameter is to be read.

NEXT_FILE (NF)

This value specifies that the tape file following the last accessed tape file will be read. If the tape is positioned at the beginning of the first volume, the first tape file will be read.

The FILE_IDENTIFIER (FI) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value is a string of 1 to 17 characters that specifies a file identifier. Each tape file on a multifile set has a unique file identifier. If the FILE_SET_POSITION parameter does not have the FILE_IDENTIFIER_POSITION value, the FILE_IDENTIFIER parameter is ignored.

If you specify the FILE_IDENTIFIER_POSITION value for the FILE_SET_POSITION parameter, the FILE_IDENTIFIER parameter must have a value. If you omit the FILE_IDENTIFIER parameter, a fatal diagnostic is issued.

The FILE_SEQUENCE_NUMBER (FSN) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value is an unsigned integer in the range 1 through 9999 that specifies the numeric position of a tape file on a multifile set. If the FILE_SET_POSITION parameter is not set to FILE_SEQUENCE_POSITION, the FILE_SEQUENCE_NUMBER parameter value is ignored.

If you specify the FILE_SEQUENCE_POSITION value for the FILE_SET_POSITION parameter, the FILE_SEQUENCE_NUMBER parameter must have a value. If you omit the FILE_SEQUENCE_NUMBER parameter, a fatal diagnostic is issued.

The GENERATION_NUMBER (GN) parameter is optional. It is for labeled tapes, and it is ignored if you specify it for an unlabeled tape. Its value, an unsigned integer in the range 1 through 9999, identifies a specific revision of the tape file named by the FILE_IDENTIFIER parameter. If the FILE_SET_POSITION parameter has the FILE_IDENTIFIER_POSITION value, and the GENERATION_NUMBER parameter is omitted, then the GENERATION_NUMBER parameter value is set to one.

If the FILE_SET_POSITION parameter does not have the FILE_IDENTIFIER_POSITION value, the GENERATION_NUMBER parameter is ignored.

The INTERNAL_CODE (IC) parameter is optional. It specifies the character set of the data on the tape volume. If you omit the parameter, its value is left at its previous setting. The INTERNAL_CODE parameter can have the following values:

AS6 6/12-bit ASCII

AS8 8/12-bit ASCII

D63 63-character display code

D64 64-character display code

The CHARACTER_CONVERSION (CC) parameter is optional. Its boolean value specifies whether or not file data is to be converted to or from the character set specified by the INTERNAL_CODE parameter. If you omit the CHARACTER_CONVERSION parameter, its value is left at its previous setting.

Of the tape file migration methods, only FMA and FMU automatically do character conversion in addition to any conversion specified by the CHARACTER_CONVERSION parameter value.

To obtain a properly migrated tape file, you usually want to set the CHARACTER_CONVERSION parameter to:

FALSE if you use FMA or FMU to migrate. Otherwise you convert your tape file data twice.

TRUE if you use any other tape file migration method. Otherwise, you do not convert your tape file data at all.

The BLOCK_TYPE (BT) parameter is optional. It specifies the block type of the NOS input tape file. If you omit this parameter, its value is left at its previous setting. Its value can be either of the following:

| | |
|---|---|
| INTERNAL (I) | Internal blocking |
| CHARACTER_COUNT (CC) | Character count blocking |

The RECORD_TYPE (RT) parameter is optional. Its value specifies the record type of the NOS tape file. If you omit this parameter, its value is left at its previous setting. Its value can be any of the following:

| | |
|---|---|
| CONTROL_WORD (CW or W) | Control word |
| FIXED_LENGTH (FL or F) | Fixed length |
| SYSTEM_RECORD (SR or S) | System record |
| ZERO_BYTE (ZB or Z) | Zero-byte |

The MAXIMUM_BLOCK_LENGTH (MAXBL or MBL) parameter is optional. Its value is an unsigned integer that specifies the maximum length in 6-bit bytes of a block in the NOS tape file. The system maximum for this parameter is 2,147,483,615. If you omit this parameter, its value is left at its previous setting.

The MAXIMUM_RECORD_LENGTH (MAXRL or MRL) parameter is optional. Its value is an unsigned integer that specifies the maximum length in 6-bit bytes of a record in the NOS tape file. The system maximum for this parameter is 4,398,046,511,103. If you omit this parameter, its value is left at its previous setting.

The TAPE_FORMAT (TF) parameter is optional. It specifies the tape format of the NOS tape file. If you omit this parameter, is value is left at its previous setting. The possible values for this parameter are:

| | |
|---|---|
| NOS_INTERNAL (NI or I) | Internal, NOS default tape format |
| STRANGER (S) | Stranger |
| LONG_STRANGER (LS or L) | Long block stranger |

The STATUS variable is optional. It is the standard SCL status variable. Refer to chapter 2 for more information.

### CHANGE_170_REQUEST Example

The example supposes that you are reading five NOS tape files from the same multifile set on a labeled tape with volume serial number CMPN. The files share the following characteristics:

Referenced by NOS/VE temporary file (FILE=MULTI_FILES)

Character data in the 64-character display code character set, the default (INTERNAL_CODE=D64)

Default character data conversion (CHARACTER_CONVERSION=FALSE)

Internal tape format, the default (TAPE_FORMAT=NOS_INTERNAL)

### CHA1R Example, First File

The first NOS tape file to be read has the default values for the BLOCK_TYPE, RECORD_TYPE, MAXIMUM_BLOCK_LENGTH, and MAXIMUM_RECORD_LENGTH parameters. It is the first tape file on the multifile set. The following command associates MULTI_FILES with this first NOS tape file:

```
/create_170_request  file=multi_files ..
.. /external_vsn='cmpn' ..
.. /file_set_position=beginning_of_set ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /label_type=standard ..
.. /tape_format=nos_internal
```

By omitting the parameters whose values were defaults, you can shorten the command to:

```
/create_170_request  file=multi_files ..
.. /external_vsn='cmpn' ..
.. /file_set_position=beginning_of_set
```

NOS/VE directs the system operator to mount the volume with external VSN CMPN. Because the tape is an ANSI labeled tape, the system verifies that the VSN recorded internally on the ANSI VOL1 label is CMPN.

### CHA1R Example, Second File

The second NOS tape file to be read is the second (and next) file in the multifile set (FILE_SET_POSITION=NEXT_FILE). It also has the default values for the BLOCK_TYPE, RECORD_TYPE, MAXIMUM_BLOCK_LENGTH, and MAXIMUM_RECORD_LENGTH parameters.

The command to change MULTI_FILES to reference this second NOS tape file is:

```
/change_170_request  file=multi_files ..
.. /file_set_position=next_file ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=nos_internal
```

If you omit the parameters whose values have not changed, you can shorten the command to:

```
/change_170_request  file=multi_files ..
.. /file_set_position=next_file
```

Note that the BLOCK_TYPE parameter value stays at INTERNAL, even though it is omitted. The BLOCK_TYPE parameter is one of the parameters that keeps its old value if you omit the parameter in a CHANGE_170_REQUEST command.

You do not have to include the FILE_SET_POSITION parameter, although its value has changed. The FILE_SET_POSITION parameter is one of the two parameters that takes on its default value if you omit the parameter in a CHANGE_170_REQUEST command. So, if you omit the FILE_SET_POSITION parameter in this example, the parameter value becomes the default NEXT_FILE for a labeled tape, and the command becomes:

```
/change_170_request  file=multi_files
```

### CHA1R Example, Third File

The third NOS tape file to be read is the sixth file in the multifile set. This means that it is found by its position in the multifile set (FILE_SET_POSITION=FILE_SEQUENCE_POSITION), and that you set the FILE_SEQUENCE_NUMBER parameter to 6. Also, the file has:

Character count blocking (BLOCK_TYPE=CHARACTER_COUNT)

Zero-byte record type (RECORD_TYPE=ZERO_BYTE)

The values of the file's MAXIMUM_BLOCK_LENGTH and MAXIMUM_RECORD_LENGTH parameters are the defaults.

The command to change MULTI_FILES to reference this third NOS tape file to be read is:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=6 ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=nos_internal
```

If you omit the parameters whose values have not changed, you can shorten the command to:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=6 ..
.. /block_type=character_count ..
.. /record_type=zero_byte
```

### CHA1R Example, Fourth File

The fourth NOS tape file to be read has the file identifier R2D2. This means that it is found in the multifile set from its file identifier (FILE_SET_POSITION=FILE_IDENTIFIER_POSITION), and that you set the FILE_IDENTIFIER parameter to R2D2. Also, the file has:

Character count blocking (BLOCK_TYPE=CHARACTER_COUNT)

Zero-byte record type (RECORD_TYPE=ZERO_BYTE)

Maximum block length of 2000 (MAXIMUM_BLOCK_LENGTH=2000)

Maximum record length of 200 (MAXIMUM_RECORD_LENGTH=200)

The command to change MULTI_FILES to reference this fourth NOS tape file to be read is:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='r2d2' ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=2000 ..
.. /maximum_record_length=200 ..
.. /tape_format=nos_internal
```

If you omit the parameters whose values have not changed, you can shorten the command to:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='r2d2' ..
.. /maximum_block_length=2000 ..
.. /maximum_record_length=200
```

Note that the values for the BLOCK_TYPE and RECORD_TYPE parameters stay at CHARACTER_COUNT and ZERO_BYTE, respectively, when you omit the parameters.

### CHA1R Example, Fifth File

The fifth NOS tape file to be read is the eighth generation of the file with identifier C3PO. This means that:

It is found in the multifile set from its file identifier (FILE_SET_POSITION= FILE_IDENTIFIER_POSITION).

You set the FILE_IDENTIFIER parameter to C3PO. (FILE_IDENTIFIER=C3PO).

You set the GENERATION_NUMBER parameter to 8. (GENERATION_NUMBER=8).

Also, the file has:

Character count blocking (BLOCK_TYPE=CHARACTER_COUNT)

Zero-byte record type (RECORD_TYPE=ZERO_BYTE)

The NOS default maximum block length and maximum record length

The command to change MULTI_FILES to reference this fifth NOS tape file to be read is:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='c3po' ..
.. /generation_number=8 ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=nos_internal
```

If you omit the parameters whose values have not changed, you can shorten the command to:

```
/change_170_request  file=multi_files ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='c3po' ..
.. /generation_number=8 ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120
```

Note that you must keep the FILE_SET_POSITION parameter, even though its value has not changed. If you omit the parameter, its value defaults to NEXT_FILE. Also, you must include the MAXIMUM_BLOCK_LENGTH and MAXIMUM_RECORD_LENGTH parameters to return their values to the NOS defaults. If you omit the parameters, their values remain at 2000 and 200, respectively.

## DISPLAY_TAPE_LABEL_ATTRIBUTES

The DISPLAY_TAPE_LABEL_ATTRIBUTES command references a temporary NOS/VE file associated with an ANSI labeled tape file. From that NOS/VE temporary file, the DISPLAY_TAPE_LABEL_ATTRIBUTES command displays the current tape label attribute information about the ANSI tape file.

If the ANSI tape file is a NOS tape file, the following is true:

The DISPLAY_TAPE_LABEL_ATTRIBUTES command displays most of the description of the NOS tape file currently associated with the NOS/VE temporary file. This description has been created by the CREATE_170_REQUEST and CHANGE_170_REQUEST commands.

The value displayed for BLOCK_TYPE is always USER_SPECIFIED (US). It is not whatever you set it to: INTERNAL or CHARACTER_COUNT.

The value displayed for CHARACTER_SET is always ASCII. It is not whatever you set INTERNAL_CODE to: AS6, AS8, D63, or D64.

The value displayed for the MAXIMUM_RECORD_LENGTH parameter is a length in 8-bit bytes. You previously set this value as a length in 6-bit bytes in a CREATE_170_REQUEST or CHANGE_170_REQUEST command. For example, your setting the MAXIMUM_RECORD_LENGTH parameter to 80 in the CREATE_170_REQUEST command says that the maximum record length is 80 6-bit bytes or 480 bits. The DISPLAY_TAPE_LABEL_ATTRIBUTES command gives the MAXIMUM_RECORD_LENGTH value of 480 bits in 8-bit bytes, namely 60.

The value displayed for the MAXIMUM_BLOCK_LENGTH parameter behaves much like the value displayed for the MAXIMUM_RECORD_LENGTH parameter. In addition, if you set the TAPE_FORMAT parameter value to NOS_INTERNAL, 8 is added to the value displayed for the MAXIMUM_BLOCK_LENGTH parameter. For example, your setting the TAPE_FORMAT parameter to NOS_INTERNAL and the MAXIMUM_BLOCK_LENGTH parameter to 400 in the CREATE_170_REQUEST command says that the maximum block length is 400 6-bit bytes or 2400 bits. The DISPLAY_TAPE_LABEL_ATTRIBUTES command displays the sum, 308, of 8 plus the MAXIMUM_BLOCK_LENGTH value of 2400 bits in 8-bit bytes, namely 300.

RECORD_TYPE is always UNDEFINED (U). It is not whatever you set it to: CONTROL_WORD, FIXED_LENGTH, SYSTEM_RECORD, or ZERO_BYTE.

An error occurs if a CREATE_170_REQUEST command has not associated the temporary NOS/VE file with a NOS tape file.

The rest of the discussion in this manual about the DISPLAY_TAPE_LABEL_ATTRIBUTES command deals with how it displays information about a NOS tape file associated with a NOS/VE temporary file by a CREATE_170_REQUEST command. See the SCL System Interface Usage manual for the complete description of the DISPLAY_TAPE_LABEL_ATTRIBUTES command.

### DISPLAY_TAPE_LABEL_ATTRIBUTES Format

The DISPLAY_TAPE_LABEL_ATTRIBUTES command has the following format:

```
DISPLAY_TAPE_LABEL_ATTRIBUTES or
DISTLA
    FILE=file
    DISPLAY_OPTIONS=list of keyword
    OUTPUT=file
    STATUS=status variable
```

The FILE (F) parameter is required. It specifies the name of the NOS/VE temporary file associated with the NOS tape file for which tape label attribute information is to be displayed.

The DISPLAY_OPTIONS (DO) parameter is optional. It specifies the tape label attribute information to be displayed. If you omit this parameter, the value ALL is assumed. The following discussion of the DISPLAY_OPTIONS values is in terms of the tape label attributes of a NOS tape file:

| | |
|---|---|
| BLOCK_TYPE (BT) | Displays USER_SPECIFIED (US) |
| BUFFER_OFFSET (BO) | Displays a zero |
| CHARACTER_CONVERSION (CC) | Displays the CHARACTER_CONVERSION value you set as YES or NO |
| CHARACTER_SET (CS) | Displays ASCII |
| CREATION_DATE (CD) | Displays the date that NOS created the NOS tape file |
| EXPIRATION_DATE (ED) | Displays the expiration date that NOS set for the NOS tape file |
| FILE_ACCESSIBILITY_CODE (FAC) | Displays the FILE_ACCESSIBILITY_CODE value that NOS chose for the NOS tape file. NOS/VE ignores this tape label attribute. |
| FILE_IDENTIFIER (FI) | Displays the FILE_IDENTIFIER value you set |
| FILE_SEQUENCE_NUMBER (FSN) | Displays the FILE_SEQUENCE_NUMBER value you set |
| FILE_SET_IDENTIFIER (FSI) | Displays the FILE_SET_IDENTIFIER value that NOS wrote for the tape volume. |
| FILE_SET_POSITION (FSP) | Displays the FILE_SET_POSITION value you set |
| GENERATION_NUMBER (GN) | Displays the GENERATION_NUMBER value you set |
| GENERATION_VERSION_NUMBER (GVN) | Displays the GENERATION_VERSION_NUMBER value that NOS chose for the NOS tape file. NOS/VE ignores this tape label attribute. |
| MAXIMUM_BLOCK_LENGTH (MAXBL) | Displays the MAXIMUM_BLOCK_LENGTH value you previously set in 6-bit bytes as a length in 8-bit bytes |
| MAXIMUM_RECORD_LENGTH (MAXRL) | Displays the MAXIMUM_RECORD_LENGTH value you previously set in 6-bit bytes as a length in 8-bit bytes |
| PADDING_CHARACTER (PC) | Displays a space |
| RECORD_TYPE (RT) | Displays UNDEFINED (U) |
| REWRITE_LABELS (RL) | Displays NO |

| | |
|---|---|
| ALL | Displays all of the above values |
| SOURCE | Displays the origin of the value of each tape label attribute specified in the DISPLAY_OPTIONS parameter |
| CURRENT_FILE | Displays the value from the currently accessed tape file of each tape label attribute specified in the DISPLAY_OPTIONS parameter. Omission causes NEXT_FILE to be used. |
| NEXT_FILE | Displays the value from the next tape file to be accessed of each tape label attribute specifiedin the DISPLAY_OPTIONS parameter. |

The OUTPUT (O) parameter is optional. It specifies a file to which the display information is written. If this parameter is omitted, $OUTPUT is assumed.

The STATUS variable is optional. It is the standard SCL status variable. Refer to chapter 2 for more information.

### DISPLAY_TAPE_LABEL_ATTRIBUTES Examples

The following DISPLAY_TAPE_LABEL_ATTRIBUTES examples are taken from previous examples shown for the CREATE_170_REQUEST and CHANGE_170_REQUEST commands. All the examples assume that the NOS tape file was created on 86-10-31 with an expiration date of 87-04-01.

#### DISTLA, File at Beginning_of_Set

The NOS tape file is the first tape file on the multifile set. The following command associates MULTI_FILES with this NOS tape file:

```
/create_170_request  file=multi_files ..
.. /external_vsn='cmpn' ..
.. /recorded_vsn='3907' ..
.. /file_set_position=beginning_of_set ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /label_type=standard ..
.. /tape_format=nos_internal
```

To display information about this NOS tape file to be accessed at the beginning of the multifile set, enter:

```
/display_tape_label_attributes  file=multi_files ..
.. /display_options=(all, next_file)
```

The following is displayed:

```
Block_Type                    : user_specified
Buffer_Offset                 : 0
Character_Conversion          : no
Character_Set                 : ascii
Creation_Date                 : unknown
Expiration_Date               : unknown
File_Accessibility_Code       : ' '
File_Identifier               : ' '
File_Sequence_Number          : 1
File_Set_Identifier           : ' '
File_Set_Position             : beginning_of_set
Generation_Number             : 1
Generation_Version_Number     : 0
Maximum_Block_Length          : 3848
Maximum_Record_Length         : 3840
Padding_Character             : ' '
Record_Type                   : undefined
Rewrite_Labels                : no
```

The values of the following tape label attributes are unknown and are displayed as such, because the NOS tape file has not yet been accessed:

```
CREATION_DATE
EXPIRATION_DATE
FILE_ACCESSIBILITY_CODE
FILE_SET_IDENTIFIER
GENERATION_VERSION_NUMBER
```

Note that the MAXIMUM_RECORD_LENGTH parameter value is displayed as 3840 8-bit bytes, which are equivalent to the 5120 6-bit bytes specified in the CREATE_170_REQUEST command. The MAXIMUM_BLOCK_LENGTH parameter value is displayed as 3848, the sum of 8 and the maximum block length in 8-bit bytes.

After the NOS tape file has been accessed, executing

```
/display_tape_label_attributes  file=multi_files ..
.. /display_options=(all, current_file)
```

results in the following display:

```
Block_Type                    : user_specified
Buffer_Offset                 : 0
Character_Conversion          : no
Character_Set                 : ascii
Creation_Date                 : 86-10-31
Expiration_Date               : 87-04-01
File_Accessibility_Code       : ' '
File_Identifier               : ' '
File_Sequence_Number          : 1
File_Set_Identifier           : '3907'
File_Set_Position             : beginning_of_set
Generation_Number             : 1
Generation_Version_Number     : 0
Maximum_Block_Length          : 3848
Maximum_Record_Length         : 3840
Padding_Character             : ' '
Record_Type                   : undefined
Rewrite_Labels                : no
```

The actual values of three tape label attributes are now known and are not the defaults:

```
Creation_Date                : 86-10-31
Expiration_Date              : 87-04-01
File_Set_Identifier          : '3907'
```

This example shows the usual FILE_SET_IDENTIFIER parameter value, namely the RECORDED_VSN parameter value.

### DISTLA, File is Next_File

This example assumes that the NOS tape file in the previous example has been accessed and that you now wish to access the next tape file. The command to change MULTI_FILES to reference this second NOS tape file is:

```
/change_170_request  file=multi_files ..
.. /file_set_position=next_file ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=nos_internal
```

To display information about the next NOS tape file in the multifile set, enter:

```
/display_tape_label_attributes  file=multi_files ..
.. /display_options=all
```

You do not need to mention the NEXT_FILE value for the DISPLAY_OPTIONS parameter, because NEXT_FILE is the default.

The following is displayed before the second NOS tape file is accessed:

```
Block_Type                   : user_specified
Buffer_Offset                : 0
Character_Conversion         : no
Character_Set                : ascii
Creation_Date                : unknown
Expiration_Date              : unknown
File_Accessibility_Code      : ' '
File_Identifier              : ' '
File_Sequence_Number         : 2
File_Set_Identifier          : ' '
File_Set_Position            : next_file
Generation_Number            : 1
Generation_Version_Number    : 0
Maximum_Block_Length         : 3848
Maximum_Record_Length        : 3840
Padding_Character            : ' '
Record_Type                  : undefined
Rewrite_Labels               : no
```

### DISTLA, Fifth File in Multifile Set

The NOS tape file in this example is the fifth file in a multifile set. The command to associate the NOS/VE temporary file FILE_5 with this NOS tape file is:

```
/create_170_request  file=file_5 ..
.. /recorded_vsn='7110' ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=5 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=stranger
```

To display information about this NOS tape file that is fifth in the multifile set, enter:

```
/display_tape_label_attributes  file=file_5 ..
.. /display_options=all
```

The following is displayed before the NOS tape file is accessed:

```
Block_Type                    : user_specified
Buffer_Offset                 : 0
Character_Conversion          : no
Character_Set                 : ascii
Creation_Date                 : unknown
Expiration_Date               : unknown
File_Accessibility_Code       : ' '
File_Identifier               : ' '
File_Sequence_Number          : 5
File_Set_Identifier           : ' '
File_Set_Position             : file_sequence_position
Generation_Number             : 1
Generation_Version_Number     : 0
Maximum_Block_Length          : 3840
Maximum_Record_Length         : 3840
Padding_Character             : ' '
Record_Type                   : undefined
Rewrite_Labels                : no
```

Note that the value of the MAXIMUM_BLOCK_LENGTH parameter is 3840, because the TAPE_FORMAT parameter has the value, STRANGER, not NOS_INTERNAL.

### DISTLA, File Identifier R2D2

The NOS tape file in this example has the file identifier R2D2 and happens to be the 29th file in a multifile set. Also, the file has:

Character count blocking (BLOCK_TYPE=CHARACTER_COUNT)

Zero-byte record type (RECORD_TYPE=ZERO_BYTE)

Maximum block length of 2000 (MAXIMUM_BLOCK_LENGTH=2000)

Maximum record length of 200 (MAXIMUM_RECORD_LENGTH=200)

The command to associate the temporary NOS/VE file ANDROID with the NOS tape file is:

```
/create_170_request  file=android ..
.. /external_vsn='3193' ..
.. /recorded_vsn='3907' ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='r2d2' ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=2000 ..
.. /maximum_record_length=200 ..
.. /tape_format=nos_internal
```

To display information about this NOS tape file identified as R2D2, enter:

```
/display_tape_label_attributes  file=android ..
.. /display_options=all
```

The following is displayed before the NOS tape file identified as R2D2 is accessed:

```
Block_Type                    : user_specified
Buffer_Offset                 : 0
Character_Conversion          : no
Character_Set                 : ascii
Creation_Date                 : unknown
Expiration_Date               : unknown
File_Accessibility_Code       : ' '
File_Identifier               : 'r2d2'
File_Sequence_Number          : 1
File_Set_Identifier           : ' '
File_Set_Position             : file_identifier_position
Generation_Number             : 1
Generation_Version_Number     : 0
Maximum_Block_Length          : 1508
Maximum_Record_Length         : 150
Padding_Character             : ' '
Record_Type                   : undefined
Rewrite_Labels                : no
```

Note that the FILE_SEQUENCE_NUMBER parameter is displayed with its default value of 1. This is because the NOS tape file has not yet been accessed, and the fact that the file is the 29th in the multifile set is not yet known.

Also note how the MAXIMUM_BLOCK_LENGTH and MAXIMUM_RECORD_LENGTH parameter values of 2000 and 200, respectively, in the CREATE_170_REQUEST command are displayed as 1508 and 150, respectively, by the DISPLAY_TAPE_LABEL_ATTRIBUTES command.

### DISTLA, Eighth Generation File

The NOS tape file in this example is the eighth generation of the file with identifier C3PO. The command to change ANDROID to reference this NOS tape file is:

```
/change_170_request  file=android ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='c3po' ..
.. /generation_number=8 ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=5120 ..
.. /tape_format=nos_internal
```

To display information about this NOS tape file identified as generation 8 of C3PO, again enter:

```
/display_tape_label_attributes  file=android ..
.. /display_options=all
```

The following is displayed before the NOS tape file identified as C3PO is accessed:

```
Block_Type                    : user_specified
Buffer_Offset                 : 0
Character_Conversion          : no
Character_Set                 : ascii
Creation_Date                 : unknown
Expiration_Date               : unknown
File_Accessibility_Code       : ' '
File_Identifier               : 'c3po'
File_Sequence_Number          : 1
File_Set_Identifier           : ' '
File_Set_Position             : file_identifier_position
Generation_Number             : 8
Generation_Version_Number     : 0
Maximum_Block_Length          : 3848
Maximum_Record_Length         : 3840
Padding_Character             : ' '
Record_Type                   : undefined
Rewrite_Labels                : no
```

## DETACH_FILE

The DETACH_FILE command deletes/detaches one or more temporary files from the $LOCAL catalog. If the temporary file references a tape file, the physical drive is released. If the temporary file is associated with a NOS tape file, the access to that NOS tape file is ended.

### DETACH_FILE Format

The DETACH_FILE command has the following format:

```
DETACH_FILE  or
DETF
    FILE=list of file
    STATUS=status variable
```

The FILE (FILES or F) parameter is required. It specifies the file or files to be detached.

The STATUS variable is optional. It is the standard SCL status variable. Refer to chapter 2 for more information.

### DETACH_FILE Example

Suppose that the following command associates the NOS/VE temporary file with a NOS tape file:

```
/create_170_request  file=one_file ..
.. /external_vsn='3193'
```

To stop the access to the NOS tape file, enter:

```
/detach_file  file=one_file
```

## Character Data Files With Tape Migration Commands

You can use FMA to migrate NOS character data files. Suppose that you wish to migrate a NOS character data tape file, with tape file identifier NOSCHAR, to the NOS/VE permanent file $USER.NOSVE_CHARACTER. NOSCHAR has the following characteristics:

The tape file is on tape volume having external VSN 5811.

The character data in the file is in the 64-character display code character set.

The character data is not to be converted because FMA does the conversion.

Character count block type

Zero-byte record type

Maximum block length of 1800 bytes

Maximum record length of 120 bytes

Internal tape format, the NOS default tape format

LGO holds the binary code of a NOS/VE FORTRAN program that uses the tape file as input.

To perform the migration, first enter the command to associate the NOS/VE temporary file ABC with the NOS tape file to be migrated:

```
/create_170_request  file=abc ..
.. /external_vsn='5811' ..
.. /file_set_position=file_identifier_position ..
.. /file_identifier='noschar' ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_block_length=1800 ..
.. /maximum_record_length=120 ..
.. /label_type=standard ..
.. /tape_format=nos_internal
```

To perform the actual migration using FMA, enter:

```
/create_file  file=$user.nosve_character

/set_file_attributes  file=$user.nosve_character ..
.. /block_type=system_specified ..
.. /record_type=variable

/open_file_migration_aid

FA/execute_migration_task ..
    migration_files=((abc  $user.nosve_character  c170_to_c180)) ..
    file=lgo

FA/close_environment
```

## Binary Data Files With Tape Migration Commands

If the tape file to be migrated has binary data on it, the data conversions are more complex, and you might need to use FMU to migrate.

For example, suppose the fifth file on the tape is to be migrated to the NOS/VE permanent file $USER.NOSVE_BINARY. The tape file has the following characteristics:

The tape file is on tape volume with external VSN U1234.

The tape file is the 5th file on the tape volume.

The character data in the file is in the 64-character display code character set.

The character data is not to be converted because FMU does the conversion.

Internal blocking type

Control word record type

Maximum block length of 5120 bytes

Maximum record length of 220 bytes

Internal tape format, the NOS default tape format

Each record contains 10 integers, followed by 10 real numbers, followed by a 20-character string.

To perform the migration using FMU, first enter the CREATE_170_REQUEST command to associate the temporary NOS/VE file TAPE1 with the NOS tape file to be migrated:

```
/create_170_request  file=tape1 ..
.. /external_vsn='U1234' ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=5 ..
.. /internal_code=d64 ..
.. /character_conversion=false ..
.. /block_type=internal ..
.. /record_type=control_word ..
.. /maximum_block_length=5120 ..
.. /maximum_record_length=220 ..
.. /label_type=standard ..
.. /tape_format=nos_internal
```

You can shorten the CREATE_170_REQUEST command by using defaults:

```
/create_170_request  file=tape1 ..
.. /external_vsn='U1234' ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=5 ..
.. /maximum_record_length=220
```

Next you enter the commands to:

- Create the NOS/VE file $USER.NOSVE_BINARY to migrate the NOS tape file to
- Set the block type and record type for $USER.NOSVE_BINARY
- Create the directives file for FMU
- Execute FMU, specifying LIST=OUTPUT to see the FMU output displayed on your terminal

```
/create_file  file=$user.nosve_binary

/set_file_attributes  file=$user.nosve_binary ..
.. /block_type=system_specified ..
.. /record_type=variable

/collect text  output=dir_file
ct? set_input_attributes  file=tape1  machine_format=c170
ct? set_output_attributes  file=nosve_binary
ct? create_output_record  file=nosve_binary
ct?     for integers=1 to 10 do
ct?         i[,8] = i[,10]
ct?     forend
ct?     for reals=1 to 10 do
ct?         f[,8] = f[,10]
ct?     forend
ct?     a[,20]
ct? create_output_record_end
ct? **

/file_management_utility  directives=dir_file ..
.. /list=output
```

## Multifile Sets With Tape Migration Commands

You can use the tape migration commands to migrate NOS tape files in succession from the same multifile set. You can associate the same temporary NOS/VE file with each of the tape files being migrated. The contents of the temporary file depend on which tape file the temporary file is referencing at the time.

The following example shows how you enter tape migration commands so that the temporary NOS/VE file correctly describes the tape file being migrated at the time. The example is of migrating the first and fourth tape files from the same multifile set on a labeled tape.

FMA is to migrate the first tape file of a multifile set to the NOS/VE permanent file $USER.NOSVE_BIN. The tape file has the following characteristics:

The tape file is on tape volume with external VSN 3801.

The tape file is the first file on the multifile set.

The character data in the file is in the 64-character display code character set.

The character data is not to be converted because FMA does the conversion.

Internal blocking type

Control word record type

Maximum block length of 3000 bytes

Maximum record length of 200 bytes

Internal tape format, the NOS default tape format

LGO holds the binary code of a NOS/VE FORTRAN program that uses the tape file as input.

The following CREATE_170_REQUEST command associates the temporary NOS/VE file SIMUL with the tape file to be migrated:

```
/create_170_request  file=simul ..
.. /external_vsn='3801' ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=1 ..
.. /character_conversion=false ..
.. /maximum_block_length=3000 ..
.. /maximum_record_length=200
```

To perform the actual migration using FMA, enter:

```
/create_file  file=$user.nosve_bin

set_file_attributes  file=$user.nosve_bin ..
.. /block_type=system_specified ..
.. /record_type=variable

/open_file_migration_aid

FA/execute_migration_task ..
    migration_files=((simul  $user.nosve_bin  c170_to_c180)) ..
    file=lgo

FA/close_environment
```

Next, FMU is to migrate the fourth file of a multifile set to the NOS/VE byte addressable permanent file $USER.NOSVE_AIRPORT. The tape file has the following characteristics:

The tape file is on tape volume with external VSN 3801.

The tape file is the fourth file on the multifile set.

The character data in the file is in the 64-character display code character set.

The character data is not to be converted because FMU does the conversion.

Character count blocking type

Zero-byte record type

Maximum block length of 3000 bytes

Maximum record length of 120 bytes

Internal tape format, the NOS default tape format

The following CHANGE_170_REQUEST command changes the temporary NOS/VE file SIMUL to describe the next tape file to be migrated:

```
/change_170_request  file=simul ..
.. /file_set_position=file_sequence_position ..
.. /file_sequence_number=4 ..
.. /block_type=character_count ..
.. /record_type=zero_byte ..
.. /maximum_record_length=120
```

The CHANGE_170_REQUEST command has to repeat the FILE_SET_POSITION parameter so that it does not default to NEXT_FILE. The command contains the following parameters because their values have changed:

FILE_SEQUENCE_NUMBER

BLOCK_TYPE

RECORD_TYPE

MAXIMUM_RECORD_LENGTH

To perform the actual migration using FMU enter:

```
/create_file  file=$user.nosve_airport

/set_file_attributes  file=$user.nosve_airport
.. /block_type=system_specified ..
.. /record_type=variable ..
        .
        .
        .

/attach_file  file=$user.nosve_airport ..
.. /access_mode=(read  write)

/collect_text  output=dir_file
ct? set_input_attributes  file=simul  machine_format=c170
ct? set_output_attributes  file=nosve_airport
ct?     **

/file_management_utility  directives=dir_file  list=output
```

Note that you use the ATTACH_FILE command to set the ACCESS_MODE file attribute for the $USER.NOSVE_AIRPORT file.

# Migrating Programs

# Approaching COBOL and FORTRAN Program Migration 13

# Approaching COBOL and FORTRAN Program Migration 13

When approaching migrating applications, you need to consider the similarities between the compilers on NOS and NOS/VE, the hardware differences in the new mainframes, and the use of dual state. This discussion presents these issues and methods of migrating programs.

## Similarities Between NOS and NOS/VE Compilers

FORTRAN for NOS/VE and FORTRAN 5 for NOS are both based on the American National Standard X3.9-1978 for the FORTRAN language. This is commonly known as FORTRAN 77 or as the ANSI77 standard.

COBOL for NOS/VE and COBOL 5 for NOS are both based on the American National Standard X3.23-1974 for COBOL. These standards provide for judging compatibility for executing program on different types of mainframes. The standards dictate basic syntax. COBOL and FORTRAN programs that use ANSI standard code on NOS should work the same on NOS/VE. However, some COBOL programs could be affected by some ANSI features not implemented in NOS/VE COBOL.

Because both NOS and NOS/VE support standard COBOL and FORTRAN, migration considerations mainly involve language syntax or characteristics that depend on the hardware and Control Data extensions. NOS/VE COBOL and FORTRAN incorporate many extensions of the NOS languages so that source code compatibility could be maintained. However, these extensions could function differently on NOS/VE.

## Major Hardware Differences

The main hardware differences are different word length and different byte length. NOS has 6-bit bytes and 60-bit words. NOS/VE has 8-bit bytes and 64-bit words. The native character set for NOS is display code. The native character set for NOS/VE is 7-bit ASCII (all 256 characters).

NOS uses mostly static (with some dynamic activity) memory with fixed maximum field length. NOS/VE uses virtual memory. Virtual memory allows you to write programs as if there were no end to computer memory. There are limits, of course, but they are beyond most programmers' needs. Virtual memory works with both data storage areas as well as executable code.

For a summary of differences and similarities between the NOS and NOS/VE systems, see appendix C.

## Using Dual State

Your new mainframe can operate with both the NOS and NOS/VE operating systems. This dual-state environment provides for migration ease and flexibility. You can pass files from one operating system to the other. This makes the task of testing and converting programs go much more smoothly than with two separate mainframes.

Also, you do not have to hurry to migrate your applications. Dual state is available and will be available for some time. Applications that are easy to convert can be migrated quickly. New applications can be developed on NOS/VE. However, applications that depend on NOS can continue to run on NOS.

# Migration Methods for COBOL and FORTRAN Programs

There are several ways to approach migrating a COBOL or FORTRAN program, depending on the migration problems in the program. Consider using one of the following methods to migrate a program.

ANSI Standard Method

NOS/VE Compiler Method for COBOL

NOS/VE FORTRAN Diagnosis

The Drawing Board

No Migration

The following paragraphs specify the steps involved in migrating a program using each of these methods.

## ANSI Standard Method

1. While running on the NOS system, compile your program with the FTN5 compiler or COBOL5 compiler ANSI parameter set to flag trivial errors in the source code. The compiler calls appear as follows:

   ```
   FTN5,I=MYSOURS,ANSI=T,B=0.

   COBOL5,I=MYSOURS,ANSI=T,EL=T,B=0.
   ```

2. Correct all errors until the program compiles successfully. The program now meets the ANSI standard.

3. For COBOL programs, check for the use of ANSI standard features not implemented in NOS/VE COBOL and recode as required. These features are listed in appendix D, Unsupported ANSI COBOL Features.

4. Check all calls to subroutines and functions (external references), and make sure that they are either supplied as a part of your source program or are acceptable externals on NOS/VE.

5. If steps 2, 3, and 4 become too work-intensive, you may want to choose the drawing board method (described later).

6. Transfer the source program to the NOS/VE state and compile it to obtain a listing from the NOS/VE COBOL or FORTRAN compiler. Check this listing carefully for more errors. There should be none.

7. Execute the program and check for execution-time logic errors.

## NOS/VE COBOL Compiler Method

1. Transfer the source program to the NOS/VE state and compile it with BASE_LANGUAGE=COBOL5 in the NOS/VE COBOL compiler call. The COBOL5 value causes the compiler to convert many COBOL 5 statements to NOS/VE COBOL.

   This COBOL5 value, however, cannot handle all conversion. Some automatically converted statements can have different processing results on NOS/VE. Also, the compiler accepts some COBOL 5 statements that can have different processing results on NOS/VE. See chapter 15, Migrating COBOL Programs, for the differences between COBOL 5 and NOS/VE COBOL.

2. Check and correct all external references not supported by NOS/VE. You might have to rewrite code to make appropriate corrections.

3. Execute the program and check for execution-time and logic errors, especially output resulting for the CDC extensions mentioned above.

## NOS/VE FORTRAN Diagnosis Method

1. Transfer the source program to the NOS/VE state, and compile it with the NOS/VE FORTRAN compiler. This method is best for programs that have many non-ANSI FORTRAN statements such as ENCODE, DECODE, BUFFER IN, and BUFFER OUT. The compiler accepts these statements; however, the results can be different from the results on NOS.

2. Check and correct all external references not supported by NOS/VE. You might have to rewrite code to make appropriate corrections.

3. Execute the program and check for execution-time and logic errors, especially output resulting for the CDC extensions mentioned above.

## The Drawing Board Method

1. After a visual examination of the program or after unsuccessful attempts at using the preceding migration methods (ANSI Standard Method, NOS/VE Compiler Method, or NOS/VE FORTRAN Diagnosis Method), you might conclude that code type migration is too costly and time-consuming. The Drawing Board method might be best for programs that have been migrated several times in the past.

2. Reduce the application to its basic algorithm. Start writing NOS/VE COBOL or FORTRAN source code from scratch. Rewriting the program, although costly, results in a better program for NOS/VE. You can make more efficient use of the CDC extensions for NOS/VE.

## No Migration Method

1. You might decide not to migrate the program but to continue to run it on NOS. This decision is appropriate for programs that are very dependent on NOS and when none of the other migration methods can be easily accomplished. Perhaps the program depends on the NOS word and character length or uses COMPASS subroutines or system macro calls.

2. You can do a slow migration using the Drawing Board method over a period of years while continuing to process in dual state at your site.

# Migrating FORTRAN Programs 14

# Migrating FORTRAN Programs 14

This chapter describes the differences between FORTRAN 5 and NOS/VE FORTRAN, and is intended as an aid in converting programs from FORTRAN 5 to NOS/VE FORTRAN.

NOS/VE FORTRAN is designed to be compatible with FORTRAN 5. However, several language extensions have been added, and several areas of incompatibility have resulted from the new operating system and hardware. Other incompatibilities are the result of FORTRAN 5 features that are not currently supported under NOS/VE FORTRAN but for which future support is anticipated.

In some cases, language incompatibilities may require program modification; in other cases, statements using incompatible features can remain in the program but will not be processed.

The differences are divided into topics as follows:

General FORTRAN Guidelines

This topic describes common programming practices that are not compatible between the two versions of FORTRAN. These practices depend on the specific characteristics of the hardware systems used by FORTRAN, such as word length and number of characters per word.

CYBER Record Manager

This topic describes differences in subprogram calls and I/O implementation.

FORTRAN Feature Differences

This topic describes specific features for which incompatibilities exist.

## General FORTRAN Guidelines

The following programming practices have different results in NOS/VE FORTRAN and FORTRAN 5. FORTRAN 5 programs that use these practices will probably require modification before they can be successfully processed under NOS/VE FORTRAN.

Coding that depends on the internal representation of data (floating-point layout, number of characters per word, and so forth) should be checked. Because of differences in word size and internal representations, these uses nearly always require modification.

Data manipulations based on the binary representation of the data should be checked. FORTRAN 5 programs that manipulate characters as octal display-coded values or as 6-bit binary digits must be modified before being compiled and executed under NOS/VE FORTRAN.

File structure and naming conventions differ significantly under NOS/VE, and default file positioning has changed. You should check all usages that depend on any of these properties.

Code that identifies or classifies information based on the location of a specific value within a specific set of memory word bits must be modified.

On both NOS and NOS/VE, over- and under-indexed arrays are a problem. FORTRAN allows them to be created, and they yield unpredictable results when processed. NOS/VE addresses this problem with the RUNTIME_CHECKS parameter in the FORTRAN command. When you specify RUNTIME_CHECKS=R, NOS/VE puts out a run-time error message whenever an array is over- or under-indexed. Moreover, NOS/VE FORTRAN provides a feature called extensible common blocks that allows arrays in common to be overindexed without the risk of conflicting with other common blocks or data areas.

Intermixed COMPASS subprograms are not supported under NOS/VE FORTRAN. COMPASS subprograms must be replaced by equivalent FORTRAN routines or SCL commands before compilation and execution under NOS/VE FORTRAN. SCL commands provide a powerful system interface that can replace many COMPASS subprograms. Almost all SCL commands are available for use with FORTRAN. SCL commands can be executed at almost any time within a FORTRAN program. Variables set outside a FORTRAN program can be tested for use by the program. Files can be attached and file attributes set at almost any time.

## CYBER Record Manager

The capabilities provided by CYBER Record Manager (CRM) on NOS are provided by the file interface routines under NOS/VE. As with CRM, all FORTRAN I/O is performed through the file interface, and a set of FORTRAN subprogram calls provides direct communication with the file interface.

The CRM/file interface differences are discussed in topics as follows:

CRM/file interface feature differences.

Sample FORTRAN program conversions; includes converting a FORTRAN program that creates an indexed sequential file and a program that reads an indexed sequential file.

Hints (SCL versus the FORTRAN file interface)

For an introductory discussion about NOS/VE files, see chapter 10.

### CRM/File Interface Feature Differences

The file interface to NOS/VE FORTRAN supports only sequential, indexed sequential, direct access, and the new byte addressable file organizations. Only files with indexed sequential or direct access organization can be accessed by direct FORTRAN calls. NOS/VE FORTRAN also supports alternate keys. The actual key file organization is not supported. The Basic Access Methods word addressable organization has been replaced by the new addressable organization.

All uses of the CRM Advanced Access Methods subprogram calls should be checked. Under FORTRAN 5, CRM offered several features within which choices could be made. However, the file interface calls under NOS/VE offer only a subset of these features. The following discussion of the direct calls to CYBER Record Manager Advanced Access Methods Version 2 (NOS and NOS/BE) is for users converting FORTRAN 5 programs to NOS/VE FORTRAN programs.

#### Alternate Keys

The FORTRAN file interface supports alternate keys. You can create alternate keys through the CREATE_ALTERNATE_INDEXES command, which is equivalent to the NOS MIPGEN utility. NOS/VE FORTRAN also provides an RMKDEF call for FORTRAN 5 compatibility.

#### File Organization

Indexed sequential (FO=IS) and direct access (FO=DA) file organizations are the only file organizations available in the file interface to FORTRAN.

#### Record Type

The record types available are F (ANSI-fixed length), U (undefined length, the default type), and V (CDC-variable length). Advanced Access Methods Version 2 record types D, R, S, T, and Z are not available in the file interface to FORTRAN.

### File Information Table

User programs do not need to reserve 35 words for the file information table. All that is needed is one word for a pointer. If the program does reserve 35 words, only the first word is used.

Values can be stored or fetched from the file information table in standard ways, that is, CALL FILEIS, CALL FILEDA, CALL STOREF, CALL IFETCH, or the function IFETCH. Values in the file information table can only be modified through the file processing calls because the file information table is an internal table that cannot be accessed directly by a program.

Any attempt to read from the table without using IFETCH will return an undefined value. If the field being read has been stored in an unconventional manner, then the value will not be returned.

Keywords must be enclosed in apostrophes; for example, 'WSA'. Boolean forms like 3LWSA (FTN4) or L"WSA" (FTN5) are not accepted.

The following CYBER Record Manager file information fields do not have equivalents in the file interface to FORTRAN:

| | | | | | |
|---|---|---|---|---|---|
| BAL | BBH | BFF | CL | BS | BT |
| BZF | B8F | CDT | DCA | CM | CNF |
| CP | CPA | C1 | EXD | DFLG | DKI |
| EFC | EO | EOFWA | HRL | FPB | FWB |
| HB | HL | LA | LAC | IBL | IRS |
| KNE | KR | LNG | LOP5 | LBL | LCR |
| LGX | LL | MFN | MNB | LP | LT |
| LVL | LX | OVF | PC | MUL | NDX |
| NOFCP | ORG | POS | PTL | PEF | PKA |
| PM | PNO | SBF | SDS | RC | RDR |
| RMK | SB | TRC | ULP | SES | SOL |
| SPR | TL | XBS | XN | VF | VNO |
| WA | WPN | BFS | | | |

Field FL, although not applicable to file interface to FORTRAN, is recognized as a synonym of field MRL.

Other keywords in the NOS CRM Advanced Access Methods interface and their meanings for the file interface to NOS/VE FORTRAN are:

DX Data exit. There is no mechanism for data exit in NOS/VE system requests, but the FORTRAN file interface saves the subroutine address, and calls the subroutine when the appropriate file position (BOI or EOI) is returned from an access operation.

FNF Fatal error flag. To allow you to read the information with a fetch request, the FORTRAN interface maintains this information in the file information table.

EFC Error file control for NOS. Not available on NOS/VE; however, use parameter DFC for message control.

EMK Embedded keys. On NOS the default value is NO (nonembedded keys); on NOS/VE the default value is YES (embedded keys).

KA Key address. The storage area for KA should be declared in unnamed COMMON.

KP Key position. This keyword, although it has no meaning in NOS/VE Advanced Access Management (AAM), is accepted by the FORTRAN file interface as a keyword in the CALL FILEIS or CALL FILEDA statement or as a parameter in the CALL STARTM, CALL STOREF, or CALL GET statement. KP is added to KA to determine the position of the key. If you include a value for KP on a CALL PUT statement, set it to 0.

KL Key length. Specify the number of characters from 1 to 255. For files with embedded keys, KL cannot be greater than the minimum record length (MRL). For integer keys used with FORTRAN, KL is 1 to 8, and you should usually specify 8 for the key length to define an 8-byte (1-word) key.

KT Key type. Conversion is usually necessary except for integer (KT=I). Three types are allowed for NOS/VE FORTRAN:

U Uncollated-character keys can be from 1 to 255 characters long and are compared using the 7-bit ASCII sequence.

I Integer keys are 8 bytes long in NOS/VE FORTRAN and are compared as signed integers. The value can be -2**63 through 2**63 -1.

S Collated-character keys can be from 1 to 255 characters long and are compared using a collation table. If you specify this key type, you must supply an explicit collation table. Several predefined collation tables are available. For more information, see the discussion on Predefined Collation Tables in chapter 11.

MBL Maximum block length. We recommend that you do not set this value but use the value calculated by the file interface.

OC Open/close flag. Although NOS/VE system requests tell whether the file is opened or closed, the file information table also contains this information, so that you can read it by an IFETCH operation.

ON Old/new flag. The file information table maintains a value of ON, which can be set to OLD (default) or NEW by a FILEIS or FILEDA call. When a CALL OPENM statement is issued, the FORTRAN interface first finds out from the system whether the file already exists. If the answer to this question conflicts with the setting of ON, a fatal error occurs.

RKW Relative key word. Should be omitted on NOS/VE because RKP provides sufficient information. If RKW is specified in a CALL FILEIS or CALL FILEDA statement, the interface multiplies the value by 10 and adds it to RKP. This might be a problem because NOS/VE has a word size of 8 bytes and not 10 bytes (NOS and NOS/BE). You should visually inspect the program to ensure that the correct value is specified.

RKP Relative key position. For NOS/VE FORTRAN, RKP specifies the byte offset into each record where the primary key begins. (The value of RKP + 1 defines the location of the first character of the key. For example if RKP=3, the key begins in the 4th character of the record. Note: one character equals one byte.)

### Reserving Space for WSA

A problem may occur if your FORTRAN 5 program uses either an INTEGER or a REAL array for WSA because NOS and NOS/BE have a word size of 10 bytes (NOS/VE has a word size of 8 bytes). For example, in NOS/VE FORTRAN, coding a statement like RECORD (8) reserves only 64 characters of space (as opposed to 80 characters in FORTRAN 5), and the first time a record is read into the area, the record overwrites the next item in memory. For this reason, your NOS/VE FORTRAN program should always declare WSA as character data type.

Also, for NOS/VE, the area for WSA should be declared as extensible COMMON.

### Optimization

FORTRAN optimization (OL=HIGH) can cause unpredictable results when WSA and/or KA are not in common. If OL=HIGH is to be used, WSA and KA should be declared as COMMON.

### Embedded Keys

The default for EMK in Advanced Access Methods Version 2 is NO (nonembedded keys). The default for the file interface to FORTRAN is YES (embedded keys).

### CALL GETNR Statement

For purposes of compatibility, the CALL GETNR statement is allowed in NOS/VE FORTRAN. CALL GETNR is treated as a CALL GETN.

### CALL SEEKF Statement

The SEEKF function does not exist in the file interface to FORTRAN. If a CALL SEEKF is encountered, the FORTRAN interface copies parameters to the file information table, sets the FILE_POSITION field to end-of-information (EOI), and returns control to the program.

## Converting Sample Program FTNIS (Creates an Indexed Sequential File)

The sample program shows creating an indexed sequential file about animals and their habitats. The discussion of the example assumes that you are familiar with the FORTRAN 5/CRM interface. The creation example includes the following:

Original Data in File ANIMALS

Converting Program 5 FTNIS

Changes to Sample FORTRAN/CRM Program FTNIS

### Original Data in File ANIMALS

A sequential file about animals and their habitats provides data for the file creation program. The record item that is defined as the primary key is the animal name. The file is shown in figure 14-1. Notice that the character positions are marked.

```
1          11         21         31   36   41   46  50   <--- Position
|-blanks---|---------|-key------|----|----|----|---|
           MAMMAL     LION       1    LAND
           BIRD       DUCK       3    AIR  LAND WATER
           MAMMAL     SEAL       2    LAND WATER
           FISH       SHARK      1    WATER
           MAMMAL     WHALE      1    WATER
           BIRD       PENGUIN    2    LAND WATER
[----------header---------------][--trailer----]
```

Figure 14-1. ANIMALS File

On NOS, this file contains trailer (T-type) records. The number in character position 31 specifies the number of trailer items in the record. Each trailer item is 5 characters long. The trailer items begin in character position 36.

The file interface must know the record length when writing a record. On NOS, the FORTRAN 5 program describes the T-type record by specifying values for fields in the file information table. CRM automatically determines record length for each record written.

On NOS/VE, the records in the animal file should be considered as type U (for undefined). There are no fields in the file information table for describing trailer type records. Therefore, the file creation program provides the file interface with the record length of each record. To do so, the program reads the field that contains the trailer count, calculates the record length, and specifies record length just before writing a record with a CALL PUT statement.

### Converting Program FTNIS

The original FORTRAN 5 program that creates an indexed sequential file is shown in figure 14-2. The program converted for execution on NOS/VE is shown in figure 14-3. In the NOS/VE FORTRAN source code, changed portions appear in lowercase letters. A complete explanation of the changes follows the program and output.

```
      PROGRAM FTNIS
      IMPLICIT INTEGER (A-Z)
      DIMENSION ISFIT (35)
      CHARACTER REC*50  <---------------------- Working storage area
C
C BUILD  FILE INFORMATION TABLE
C
      CALL FILEIS (ISFIT, 'LFN', 'ISFILE', 'ORG', 'NEW',
     X 'RT', 'T', 'HL', 35, 'TL', 5, 'CP', 30, 'CL', 1,
     X 'MRL', 50, 'WSA', REC,
     X 'DP', 80, 'IP', 20, 'MBL', 600,
     X 'KT', 'S', 'KL', 10, 'RKW', 2, 'RKP', 0, 'EMK', 'YES',
     X 'KA', REC (21:30), 'KP', 0,
     X 'EFC', 3)
      CALL OPENM (ISFIT, 'NEW')
      CALL FITDMP (ISFIT)
C
C COPY SEQUENTIAL FILE TO INDEXED SEQUENTIAL
      OPEN (1, FILE = 'ANIMALS')
      REWIND 1
      PRINT 901
  100 READ (1, '(A)', END=200) REC
      CALL PUT (ISFIT)
      PRINT 902, IFETCH (ISFIT, 'FP'), IFETCH (ISFIT, 'RL'), REC
      GO TO 100
  200 CALL FITDMP (ISFIT)
      CALL CLOSEM (ISFIT)
      PRINT *, ' READ -IS- FILE'
      CALL OPENM (ISFIT, 'INPUT')
  300 CALL GETN (ISFIT)
      IF (IFETCH (ISFIT(1), 'FP') .EQ. O"100") GO TO 400
      RL = IFETCH (ISFIT, 'RL')
      PRINT 902, IFETCH (ISFIT(1), 'FP'), RL, REC (1:RL)
      GO TO 300
  400 CONTINUE
      CALL CLOSEM (ISFIT)
  901 FORMAT ('1 WRITE -IS- FILE')
  902 FORMAT (' FP = ', O4, ' RL = ', I3, ' REC = ', A)
      END
```

Figure 14-2. NOS FORTRAN 5 Program FTNIS

```
      PROGRAM FTNIS
      IMPLICIT INTEGER (A-Z)
      integer isfit
      CHARACTER REC*50
      common rec
C
C BUILD  FILE INFORMATION TABLE
C
      CALL FILEIS (ISFIT, 'LFN', 'ISFILE',
     X 'rt', 'u',
     X 'MRL', 50, 'WSA', REC,
     X 'DP', 80, 'IP', 20,
     X 'kt', 'u', 'KL', 10, 'rkp', 20, 'EMK', 'YES',
     X 'KA', REC (21:30), 'KP', 0,
     X 'dfc', 3)
      CALL OPENM (ISFIT, 'NEW')
C
C COPY SEQUENTIAL FILE TO INDEXED SEQUENTIAL
      OPEN (1, FILE = 'ANIMALS')
      REWIND 1
      PRINT 901
  100 READ (1, '(A)', END=200) REC
      read (rec (31:31), '(i1)') tc
      reclen = 35 + tc*5
      call storef (isfit, 'RL', reclen)
      CALL PUT (ISFIT)
      if (ifetch (isfit, 'es') .ne. 0) go to 400
      PRINT 902, IFETCH (ISFIT, 'FP'), IFETCH (ISFIT, 'RL'), REC
      GO TO 100
  200 CALL CLOSEM (ISFIT)
      PRINT *, ' READ -IS- FILE'
      CALL OPENM (ISFIT, 'INPUT')
      if (ifetch (isfit, 'es' ) .ne. 0) go to 400
  300 CALL GETN (ISFIT)
      IF (IFETCH (ISFIT, 'FP') .eq. 64) GO TO 400
      RL = IFETCH (ISFIT, 'RL')
      PRINT 902, IFETCH (ISFIT, 'FP'), RL, REC (1:RL)
      GO TO 300
  400 CONTINUE
      CALL CLOSEM (ISFIT)
  901 FORMAT ('1 WRITE -IS- FILE')
  902 FORMAT (' FP = ', I4, ' RL = ', I3, ' REC = ', A)
      END
```

Figure 14-3. Converted NOS/VE FORTRAN Program FTNIS

The commands to execute PROGRAM FTNIS on NOS/VE are as follows:

| | |
|---|---|
| /get_file to=animals<br>/get_file to=finis | Transfers the ANIMALS file and the FORTRAN source file FTNIS to the NOS/VE state. Assumes the files are in 6/12-bit display code format under your user name on NOS. |
| /create_file file=$user.isfile | Creates an empty permanent file ISFILE in your user catalog and attaches the file for local use with the name ISFILE. |
| /fortran input=ftnis l=list | Identifies the source file; designates that the listing be written to file LIST and (by default) the object code be written to file LGO. |
| /lgo | Executes the program. |

The execution output is shown in figure 14-4.

```
 WRITE -IS- FILE
FP = 1 RL =  40 REC =          MAMMAL    LION      1    LAND
FP = 1 RL =  50 REC =          BIRD      DUCK      3    AIR  LAND WATER
FP = 1 RL =  45 REC =          MAMMAL    SEAL      2    LAND WATER
FP = 1 RL =  40 REC =          FISH      SHARK     1    WATER
FP = 1 RL =  40 REC =          MAMMAL    WHALE     1    WATER
FP = 1 RL =  45 REC =          BIRD      PENGUIN   2    LAND WATER
-- File ISFILE : 0 DELETE_KEYs done since last open.
-- File ISFILE : 0 GET_KEYs done since last open.
-- File ISFILE : 0 GET_NEXT_KEYs done since last open.
-- File ISFILE : 6 PUT_KEYs (and PUTREPs->put) since last open.
-- File ISFILE : 0 PUTREPs done since last open.
-- File ISFILE : 0 REPLACE_KEYs (and PUTREPs->replace) since last open.

 READ -IS- FILE
FP =16 RL =  50 REC =          BIRD      DUCK      3    AIR  LAND WATER
FP =16 RL =  40 REC =          MAMMAL    LION      1    LAND
FP =16 RL =  45 REC =          BIRD      PENGUIN   2    LAND WATER
FP =16 RL =  45 REC =          MAMMAL    SEAL      2    LAND WATER
FP =16 RL =  40 REC =          FISH      SHARK     1    WATER
FP =16 RL =  40 REC =          MAMMAL    WHALE     1    WATER
-- File ISFILE : AMP$GET_NEXT_KEY has reached a file boundary : EOI.
-- File ISFILE : 0 DELETE_KEYs done since last open.
-- File ISFILE : 0 GET_KEYs done since last open.
-- File ISFILE : 6 GET_NEXT_KEYs done since last open.
-- File ISFILE : 0 PUT_KEYs (and PUTREPs->put) since last open.
-- File ISFILE : 0 PUTREPs done since last open.
-- File ISFILE : 0 REPLACE_KEYs (and PUTREPs->replace) since last open.
```

Figure 14-4. Output From NOS/VE Program FTNIS

### Changes to Sample FORTRAN/CRM Program FTNIS

Only essential changes were made to the program. For migration ease, the NOS/VE file interface recognizes some FORTRAN 5 parameters that are not applicable to NOS/VE and issues a trivial error but continues processing. The following list details the changes made to the program.

1. Redefined ISFIT. For FORTRAN 5, ISFIT is a 35-word integer array; for NOS/VE FORTRAN ISFIT, ISFIT is an integer item (a variable). In the program, the statement DIMENSION ISFIT (35) is changed to INTEGER ISFIT. The subscript notation is deleted in IFETCH (ISFIT...) function references.

2. Declared the record area (also called the working storage area) for the file as unnamed COMMON. Added the statement:

   ```
   COMMON REC
   ```

3. Changed the file information table set up in the CALL FILEIS statement. Table 14-1 lists all the fields and indicates changes.

Table 14-1. File Information Table Summary

| NOS FTN 5 | NOS/VE FTN | Comments |
|---|---|---|
| fitname | fitname | Identifies the file information table; same. |
| LFN | LFN | Local file name; same. |
| CP | n/a | Trailer count; discussed later as the trailer group. |
| CL | n/a | Count field length; discussed later as the trailer group. |
| DP | DP | Data padding; same. |
| EFC | DFC | EFC (error file control) has no NOS/VE equivalent; however, you should use DFC in a NOS/VE FORTRAN call to specify message control. |
| EMK | EMK | Embedded key; same. |
| HL | n/a | Header length; discussed later as the trailer group. |
| IP | IP | Index padding; same. |
| KL | KL | Key length; same |
| KA | KA | Key location; same. |
| KP | KP | Beginning character position of the key. Not applicable for NOS/VE FORTRAN; deleted in table; RKP provides the information. For file migration ease, this parameter can be specified. Set KP to 0 if you specify it on a CALL PUT statement. |
| KT | KT | Key type; changed from S (symbolic) to U (uncollated-character keys), which are compared using the 7-bit ASCII sequence. |
| MRL | MRL | Maximum record length; same. |
| ORG | n/a | NOS organization of new or old; omit this parameter in the NOS/VE FORTRAN program because it is not applicable. |
| RKW | n/a | Relative key word; omit RKW in the NOS/VE FORTRAN program because RKP provides sufficient information. |
| RKP | RKP | Relative key position. For NOS/VE FORTRAN, RKP specifies the byte offset into each record where the primary key begins. (The value of RKP + 1 defines the location of the first character of the key. For example if RKP=3, the key begins in the 4th character of the record. Note: one character equals one byte.) |
| RT | RT | Record type; required conversion from T (trailer) to U (undefined). For indexed sequential files, types U and V are internally equivalent to the file interface. |
| TL | n/a | Trailer length; discussed later as the trailer group. |
| WSA | WSA | Working storage area; same specification; however, in NOS/VE FORTRAN the area should be declared as unnamed common. |

4. Added statements to determine record length and store the information in the file information table. This procedure is necessary because the file information table no longer includes information about the trailer portion of the record. On NOS the following fields provided trailer group information:

   HL Header length
   TL Trailer length
   CP Trailer count
   CL Count field length

   These fields are not applicable on NOS/VE. A NOS T-type record must be converted to either of the following NOS/VE record types: undefined (U) or variable (V).

   To determine record length, the program reads the item in the record area giving the trailer count, calculates record length by adding the header portion of the record (35 characters) to the product of trailer length (5 characters) times trailer count, and stores the record length (STOREF routine). The code added is:

```
READ (REC (31:31), '(I1)') TC
RECLEN = 35 + TC*5
CALL STOREF (ISFIT, 'RL', RECLEN)
```

5. Added statement to check for an error on the CALL PUT and again after the CALL OPENM.

```
IF (IFETCH (ISFIT, 'ES') .NE. 0) GO TO 400
```

6. Deleted call to FITDMP. FITDMP is not available on NOS/VE.

7. Changed the file position value 100 octal to 64 decimal and the corresponding edit descriptor on the FORMAT statement from O to I.

## Converting Sample Program READIS (Reads an Indexed Sequential File)

The following program reads the indexed sequential animals file (ISFILE) created by program FTNIS. First, the program reads the file sequentially. Then the program requests that the user enter animal names for random reads of the file. The program terminates when the user types END. For a description of the data file, see the description of the Original Data in File ANIMALS earlier in this chapter.

The original FORTRAN 5 program for use on NOS is shown in figure 14-5. It reads the animals file called ISFILE. Note that once the file is created, the CALL FILEIS statement does not have to provide all the information specified for a creation run--just information to identify the file, the working storage area, and the key.

```
      PROGRAM READIS
      IMPLICIT INTEGER (A-Z)
      DIMENSION ISFIT (35)
      CHARACTER REC*50
      CHARACTER ISKEY*10
C
C PROVIDE FILE INFORMATION
C
      CALL FILEIS (ISFIT, 'LFN', 'ISFILE', 'ORG', 'NEW',
     X 'WSA', REC, 'KA', ISKEY, 'KP', 0)
      CALL OPENM (ISFIT, 'INPUT','R')
C
C  SEQUENTIAL READS
C
      PRINT *, ' SEQUENTIAL READ -IS- FILE'
  100 CALL GETN (ISFIT)
      IF (IFETCH (ISFIT(1), 'FP') .EQ. 0"100") GO TO 200
      RL = IFETCH (ISFIT, 'RL')
      PRINT 901, IFETCH (ISFIT(1), 'FP'), RL, REC (1:RL)
      GO TO 100
  200 CONTINUE
C
C  RANDOM READS
C
      PRINT*, ' RANDOM READ -IS- FILE'
  300 PRINT*, '  '
      PRINT*, ' ENTER ANIMAL -I.E. ''WHALE''- OR ''END'' '
      READ*, ISKEY
      IF (ISKEY .EQ. 'END') GO TO 400
      CALL GET (ISFIT)
      IF (IFETCH (ISFIT(1), 'FP') .EQ. 0"100") GO TO 400
      RL = IFETCH (ISFIT, 'RL')
      PRINT 901, IFETCH (ISFIT(1), 'FP'), RL, REC (1:RL)
      GO TO 300

  400 CALL CLOSEM (ISFIT)
  901 FORMAT (' FP = ', O4, ' RL = ', I3, ' REC = ', A)
      END
```

Figure 14-5. NOS FORTRAN 5 Program READIS

## NOS/VE FORTRAN Program READIS

The program converted for NOS/VE is shown in figure 14-6. The changed code appears in lowercase letters. Brief notations appear to the side.

A major change required specifying that the ASCII6 collating sequence (OSV$ASCII6_FOLDED listed in appendix C) be used instead of 7-bit ASCII. With ASCII in effect, the typed entry "lion" is treated differently from "LION". To make the program insensitive to uppercase and lowercase letters, and, therefore, act like a NOS FORTRAN program, the ASCII6 sequence is used.

```
      PROGRAM READIS
      IMPLICIT INTEGER (A-Z)
      integer isfit  <---------------------- Changed from 35 word array.
      CHARACTER REC*50
      CHARACTER ISKEY*10
      equivalence (rec (21:30), iskey)  <---- Equivalence for documentation.
      common rec                              Declare area in common.
C
C PROVIDE FILE INFORMATION  <---------------- Field DFC added below for error
C                                             listings.
      CALL FILEIS (ISFIT, 'LFN', 'ISFILE',
     X 'WSA', REC, 'KA', ISKEY, 'KP', 0, 'dfc', 3, 'MRL', 50)
      CALL OPENM (ISFIT, 'INPUT','R')
C
C  SEQUENTIAL READS
C
      PRINT *, ' SEQUENTIAL READ -IS- FILE'
  100 CALL GETN (ISFIT)
      IF (IFETCH (isfit, 'FP') .EQ. 64) GO TO 200
      RL = IFETCH (ISFIT, 'RL')
      PRINT 901, IFETCH (isfit, 'FP'), RL, REC (1:RL)
      GO TO 100
  200 CONTINUE
C
C  RANDOM READS
C  Change collate table from ASCII to ASCII6 so that
C  corresponding upper and lowercase letters are equal.
C
      call colseq ('ASCII6')
      PRINT*, ' RANDOM READ -IS- FILE'
  300 PRINT*, ' '
      PRINT*, ' ENTER ANIMAL -I.E. ''WHALE''- OR ''END'' '
      READ*, ISKEY
      IF (ISKEY .EQ. 'END') GO TO 400
      CALL GET (ISFIT)
      IF (IFETCH (isfit, 'FP') .EQ. 64) GO TO 400
      RL = IFETCH (isfit, 'RL')
      PRINT 901, IFETCH (isfit, 'FP'), RL, REC (1:RL)
      GO TO 300
  400 CALL CLOSEM (ISFIT)
  901 FORMAT (' FP = ', I4, ' RL = ', I3, ' REC = ', A)
      END
```

Figure 14-6. Converted NOS/VE FORTRAN Program READIS

The commands to compile and execute READIS on NOS/VE are as follows:

| | |
|---|---|
| /get_file to=readis | Transfers the file READIS to NOS/VE. Assumes that the file is in 6/12-display code format under your user name on NOS. |
| /attach_file file=$user.isfile | Assumes the indexed sequential file ISFILE is in your user catalog. If it is in someone else's catalog, specify:<br><br>ATTF .username.ISFILE |
| /fortran i=readis dc=user l=list | Identifies the source file; designates writing the listing to file LIST and (by default) writing the object code to file LGO. DC=USER required for COLSEQ call. |
| /lgo | Executes the program. |

## Hints—SCL Versus the FORTRAN File Interface

The following hints are for readers who use NOS/VE commands, such as the SET_FILE_ATTRIBUTES command, to establish file characteristics (attributes). Some system terminology about file attributes can be confusing to the person who also uses the FORTRAN file interface routines.

First, from a FORTRAN file interface user's point-of-view, all the fields in the file information table look like parameters that establish file attributes. However, in NOS/VE file interface terminology, some fields are called file attributes and other fields are called parameters. For a functional difference:

> A file attribute can be set or changed with the SET_FILE_ATTRIBUTES command. A parameter cannot be specified in the SET_FILE_ATTRIBUTES command; parameters can be set only on calls from the FORTRAN program (for example, the CALL PUT routine).

However, both file attributes and parameters can be set with the CALL FILEIS routine.

Secondly, the abbreviations for some file information fields used in the file interface calls (CALL FILEIS, CALL STOREF, and so forth) are different from those used in the SET_FILE_ATTRIBUTES (SETFA) command. If you use the SET_FILE_ATTRIBUTES command, you should be aware of these differences. (Anyone doing file conversion with the File Management Utility uses the SET_FILE_ATTRIBUTES command and should continue reading.)

The FORTRAN file interface provides SCL-style names and SCL-style abbreviations for file attributes in addition to the FORTRAN 5 like abbreviations.

For example, in the CALL FILEIS statement, maximum record length can be specified in any of the following three ways:

| | |
|---|---|
| MRL | FORTRAN abbreviation |
| $MAXIMUM_RECORD_LENGTH | SCL name for the FORTRAN file interface |
| $MAXRL | SCL abbreviation for the FORTRAN file interface |

In a SET_FILE_ATTRIBUTES command, maximum record length can be specified only as MAXIMUM_RECORD_LENGTH (or MAXRL).

Table 14-2 lists file attributes together with the three forms they can be specified in the FORTRAN file interface.

Table 14-2. Summary of File Attributes

| Attribute or Parameter Name | SCL-Style Name | SCL-Style Abbreviation | FORTRAN Abbreviation |
|---|---|---|---|
| ACCESS_MODE | $ACCESS_MODE | $AM | PD |
| AUTOMATIC_UNLOCK | $AUTOMATIC_UNLOCK | $AU | AU |
| AVERAGE_RECORD_LENGTH | $AVERAGE_RECORD_LENGTH | $ARL | ARL |
| COLLATE_TABLE_NAME | $COLLATE_TABLE_NAME | $CTN | CTN |
| COLLATE_TABLE | $COLLATE_TABLE | $CT | DCT |
| DATA_EXIT_ROUTINE | - | - | DX |
| DATA_PADDING | $DATA_PADDING | $DP | DP |
| EMBEDDED_KEY† | $EMBEDDED_KEY | $EK | EMK |
| ERROR_COUNT | $ERROR_COUNT | $EC | ECT |
| ERROR_EXIT_NAME | $ERROR_EXIT_NAME | $EEN | EXN |
| ERROR_EXIT_PROCEDURE | $ERROR_EXIT_PROCEDURE | $EEP | EX |
| ERROR_LIMIT | $ERROR_LIMIT | $EL | ERL |
| ERROR_STATUS | $ERROR_STATUS | $ES | ES |
| ESTIMATED_RECORD_COUNT | $ESTIMATED_RECORD_COUNT | $ERC | ERC |
| FATAL/NONFATAL | - | - | FNF |
| FILE_IDENTIFIER | $FILE_IDENTIFIER | $FI | - |
| FILE_ORGANIZATION | $FILE_ORGANIZATION | $FO | FO |
| FILE_POSITION | $FILE_POSITION | $FP | FP |
| FORCED_WRITE | $FORCED_WRITE | $FW | FWI |
| GET_AND_LOCK | $GET_AND_LOCK | $GAL | GAL |
| HASHING_PROCEDURE_NAME | $HASHING_PROCEDURE_NAME | $HPN | HPN |
| INDEX_LEVELS | $INDEX_LEVELS | $IL | NL |
| LEVELS_OF_INDEXING | - | - | NL |
| INDEX_PADDING | $INDEX_PADDING | $IP | IP |
| INITIAL_HOME_BLOCK_COUNT | $INITIAL_HOME_BLOCK_COUNT | IHBC | HMB |
| KEY_ADDRESS | $KEY_ADDRESS | $KA | KA |
| KEY_LENGTH | $KEY_LENGTH | $KL | KL |
| KEY_TYPE | $KEY_TYPE | $KT | KT |
| KEY_POSITION | $KEY_POSITION | $KP | KP |
| KEY_RELATION | $KEY_RELATION | $KR | KR |
| LAST_OPERATION | $LAST_OPERATION | $LO | LOP |
| LOCAL_FILE_NAME | $LOCAL_FILE_NAME | $LFN | LFN |
| LOCK_EXPIRATION_TIME | $LOCK_EXPIRATION_TIME | $LET | $LET |
| LOCK_INTERLOCK | $LOCK_INTERLOCK | $LI | LI |
| MAJOR_KEY_LENGTH | $MAJOR_KEY_LENGTH | $MKL | MKL |
| MAXIMUM_BLOCK_LENGTH | $MAXIMUM_BLOCK_LENGTH | $MAXBL | MBL |
| MAXIMUM_RECORD_LENGTH | $MAXIMUM_RECORD_LENGTH | $MAXRL | MRL |
| MESSAGE_CONTROL | $MESSAGE_CONTROL | $MC | DFC |
| MINIMUM_RECORD_LENGTH | $MINIMUM_RECORD_LENGTH | $MINRL | MNR |
| OLD/NEW | - | - | ON |
| OPEN/CLOSE | - | - | OC |
| OPEN_POSITION | $OPEN_POSITION | $OP | OF |
| RECORD_LENGTH | - | - | RL |
| RECORD_LIMIT | $RECORD_LIMIT | $RL | FLM |
| RECORD_TYPE | $RECORD_TYPE | $RT | RT |
| RECORDS_PER_BLOCK | $RECORDS_PER_BLOCK | $RPB | RB |
| RESIDUAL_SKIP_COUNT | - | - | RA |
| SKIP_COUNT | $SKIP_COUNT | $SC | SKP |
| WORKING_STORAGE_AREA | $WORKING_STORAGE_AREA | $WSA | WSA |
| WORKING_STORAGE_LENGTH | $WORKING_STORAGE_LENGTH | $WSL | WSL |

†Use the values TRUE or FALSE for an embedded key in the SETFA command; but use the values YES or NOS for an embedded key in the CALL FILEIS statement.

## FORTRAN Feature Differences

The following topics describe the specific differences and areas of incompatibility between NOS/VE FORTRAN and FORTRAN 5.

FORTRAN-callable subprogram differences

Input/Output -- Includes: Buffer I/O, O and Z editing, ENCODE/DECODE, files INPUT and OUTPUT, and the maximum length formatted records.

Loading -- Includes: Overlays, OVCAPs, segment loading, and static loading.

FORTRAN command -- Includes: Command parameters for NOS/VE FORTRAN.

Other -- Boolean data type, double precision functions, procedure communication, floating-point arithmetic, default collating sequence, FTN5 command, and the PROGRAM, SAVE, and LABEL statements.

## FORTRAN-Callable Subprogram Differences

The discussion of FORTRAN-callable subroutine differences includes the following routines: Post Mortem Dump, LABEL, GETPARM, CHECKPTX, RECOVR, DATE, TIME, CLOCK, LOCF, SECOND, 8-Bit, and permanent file routines.

This discussion also shows using the NOS/VE parameter interface subprograms, which replace the GEPPARM subroutine.

### Post Mortem Dump

The initial release of NOS/VE FORTRAN does not support the Post Mortem Dump debugging facility. The calls to PMDARRY, PMDDUMP, PMDLOAD, and PMDSTOP are provided but are ignored during compilation and execution.

### Subroutine LABEL

FORTRAN 5 subroutine LABEL is not supported by NOS/VE FORTRAN. A LABEL subroutine is provided but it performs no operation.

### 8-Bit Subroutines

The 8-bit subroutines are not supported under NOS/VE FORTRAN.

### Permanent File Subroutines

The NOS permanent file subroutines are not supported under NOS/VE FORTRAN. These are replaced by the SCL interface.

### Subroutine GETPARM

FORTRAN 5 subroutine GETPARM is replaced by the SCL interface capability under NOS/VE FORTRAN. A GETPARM subroutine is provided but it performs no operation. For information and example about the NOS/VE replacement feature, see the Using the NOS/VE Parameter Interface Subprograms, which is later in this chapter.

### Subroutines CHEKPTX and RECOVR

FORTRAN 5 subroutines CHEKPTX and RECOVR are not supported by NOS/VE FORTRAN. CHEKPTX and RECOVR subroutines are provided but perform no operation.

### DATE, TIME, and CLOCK Functions

The values returned by the NOS/VE FORTRAN DATE, TIME, and CLOCK functions have formats different from those used by FORTRAN 5. See the FORTRAN Language Definition Usage manual for the formats. The length declared for TIME or CLOCK by the CHARACTER statement must be changed to 8. (The length of DATE is still 10.)

### LOCF Function

The FORTRAN 5 LOCF function is not currently supported under NOS/VE FORTRAN.

### SECOND Function

Under NOS/VE FORTRAN, the SECOND function is supplied as a utility subprogram rather than as an intrinsic function. Thus, any FORTRAN 5 programs that declare the SECOND function in an INTRINSIC statement should be changed to declare the function in an EXTERNAL statement.

### SYSTEMC or SYSTEM Calls

FORTRAN 5 error numbers are automatically mapped into the corresponding NOS/VE FORTRAN error numbers for use with the SYSTEM or SYSTEMC calls.

## Using the NOS/VE Parameter Interface Subprograms

The NOS/VE FORTRAN parameter interface subprograms replace the FORTRAN 5 GETPARM subroutine. These subprograms provide an improved capability for NOS/VE FORTRAN in passing parameters to executing programs. This discussion describes the function of the subprograms and shows using them.

A series of interface subprograms provide for parameter verification and evaluation. The subprograms together with the values they return are:

| | |
|---|---|
| GETBVAL | Value of an SCL boolean parameter |
| GETCVAL | Value of a STRING, NAME, or FILE parameter |
| GETIVAL | Value of an SCL integer parameter |
| GETSCNT | Integer indicating the number of value sets specified for the parameter |
| GETSVAL | Values of an SCL status parameter |
| GETVCNT | The number of values in the specified value set of the specified parameter |
| GETVREF | Variable reference specified for a parameter |
| SCLKIND | String indicating the SCL kind (type) of a parameter |
| TSTPARM | Logical value .TRUE. if the specified parameter appears on the execution command. Otherwise, .FALSE. |
| TSTRANG | Logical value .TRUE. if the named value was specified on the execution command as a range. Otherwise .FALSE. (Values in the form m .. z indicate a range of values from m to z.) For a discussion of the ellipsis used to show value range for a parameter, see chapter 2. |

### Value Sets

Parameters are frequently lists and value sets. The interface subprograms in the example check and evaluate integer parameters. To understand the example, you need to understand how to specify parameters as a list of values, as a range of values, and as value sets as discussed in chapter 2. The remainder of this discussion explains more complex situations involving value sets.

A value set in a parameter is a list of values. For example:

```
LGO (1 2 3)
      |
      +----------> The list is a simple value set. A value set is enclosed in parentheses.
```

The following command specifies a more complex value set. The ellipsis indicates ranges. Each set of range indicators is considered as one item.

```
                    +----------> A value set of 3 items.
                    |
LGO (1 2 3..5 (4 5 6..9))
                 |
                 +--------------> A value set of 4 items because (4 5 6..9) is considered as 1.
```

### Interface Subprogram Example

The interface subprogram example checks and evaluates integer parameters to demonstrate the use of specific subroutines. The example consists of the NOS/VE FORTRAN program included in the SCL procedure that compiles and executes the program. The program is executed five times, with different parameters passed each time. The procedure listing, figure 14-7, is followed by the output listing produced by the program, figure 14-8. The program and program output are self-documenting. The LGO command being processed is duplicated in the program output.

```
proc ptesti
Collect_text ftninp
      program ptesti
      implicit integer (a-z)
      integer GETSCNT, GETVCNT
      logical TSTPARM, TSTRANG
C$    PARAM ('int: list 1..10 1..3 range of integer')
c
c  First make sure the parameter was there at all.
c
      if (TSTPARM('int')) then
        print *, '"Int" parameter was coded on the execution statement.'
        print *, ' '
c
c  Then get the number of value sets that were specified.
c
        numset = GETSCNT('int')
        print *, 'There are ', numset, ' value sets for int.'
c
c  Process each value set.
c
        do 100,set=1,numset
c
c  Get the number of values specified in this value set.
c
          numval = GETVCNT('int', set)
          print *, ' '
          print *, 'There are ', numval, ' values in set #',set
c
c  Process each value in the value set.
c
          do 100, val=1,numval
c
c  Get this value of this set (low value if it is a range).
c
            call GETIVAL('int',set,val,'low',ival,radix)
c
c  Give special treatment to base 10 values versus all others.
c
            if (radix .eq. 10) then
              print *, 'Low value of value #', val, ' is ', ival
            else
              print *, 'Low value of value #', val, ' is ', ival,
     +            ' (base 10), originally entered in base ',radix
              end if
```

(Continued on next page)

Figure 14-7. Procedure PTESTI, Evaluates Integer Parameters

```
+--(Continued from previous page)-----------------------------------------------+
|                                                                               |
|  c                                                                            |
|  c  If it is a range value, get the high value for the range.                 |
|  c                                                                            |
|             if (TSTRANG('int',set,val))then                                   |
|               call GETIVAL('int',set,val,'high',ival,radix)                  |
|  c                                                                            |
|  c  Give special treatment to base 10 values versus all others.               |
|  c                                                                            |
|               if (radix .eq. 10) then                                         |
|                 print *, '   and the high value is ',ival                     |
|               else                                                            |
|                 print *, '   and the high value is ',ival,                    |
|      +              ' (base 10), originally entered in base ',radix           |
|                 end if                                                        |
|               end if                                                          |
|    100    continue                                                            |
|         else                                                                  |
|           print *, '"Int" parameter was not coded.'                           |
|           end if                                                              |
|         end  <------------------------------------------- End of program      |
|  **  <--------------------------------------------------- Terminates COLLECT_TEXT
|  fortran input=ftninp binary_object=lgo                                       |
|  create_variable variable=str kind=string                                     |
|  create_file_connection standard_file=$echo file=output                       |
|  lgo                                                                          |
|  lgo 1                                                                        |
|  lgo (1 2 3)                                                                  |
|  lgo 1(16)                                                                    |
|  lgo (54(8)..62(16))                                                          |
|  lgo (1 2 3..5 (4 5 6..9))                                                    |
|  delete_file_connection standard_file=$echo file=output                       |
|  detach_file file=ftninp                                                      |
|  procend ptesti  <--------------------------------------- Terminates procedure
|                                                                               |
+-------------------------------------------------------------------------------+
```

Figure 14-7. Procedure PTESTI, Evaluates Integer Parameters

```
+-------------------------------------------------------------------------------+
|                                                                               |
|  CI lgo                                                                       |
|  "Int" parameter was not coded.                                               |
|  CI lgo 1                                                                     |
|  "Int" parameter was coded on the execution statement.                        |
|                                                                               |
|  There are 1 value sets for int.                                              |
|                                                                               |
|  There are 1 values in set #1                                                 |
|  Low value of value #1 is 1                                                   |
|                                                                               |
|  CI lgo (1 2 3)  <-------------------------------------- LGO command          |
|  "Int" parameter was coded on the execution statement.                        |
|                                                                               |
+-------------------------------------------------(Continued on next page)--+
```

Figure 14-8. Execution Output of Procedure PTESTI

```
+--(Continued from previous page)-----------------------------------------------+
|                                                                               |
|  There are 3 value sets for int.                                              |
|                                                                               |
|  There are 1 values in set #1                                                 |
|  Low value of value #1 is 1                                                   |
|                                                                               |
|  There are 1 values in set #2                                                 |
|  Low value of value #1 is 2                                                   |
|                                                                               |
|  There are 1 values in set #3                                                 |
|  Low value of value #1 is 3                                                   |
|                                                                               |
|                                                                               |
|  CI lgo 1(16)  <------------------------------------------- LGO command       |
|  "Int" parameter was coded on the execution statement.                        |
|                                                                               |
|  There are 1 value sets for int.                                              |
|                                                                               |
|  There are 1 values in set #1                                                 |
|  Low value of value #1 is 1 (base 10), originally entered in base 16          |
|                                                                               |
|                                                                               |
|  CI lgo (54(8)..62(16))  <------------------------------------ LGO command    |
|  "Int" parameter was coded on the execution statement.                        |
|                                                                               |
|  There are 1 value sets for int.                                              |
|                                                                               |
|  There are 1 values in set #1                                                 |
|  Low value of value #1 is 44 (base 10), originally entered in base 8          |
|     and the high value is 98 (base 10), originally entered in base 16         |
|                                                                               |
|                                                                               |
|  CI lgo (1 2 3..5 (4 5 6..9))  <------------------------------ LGO command    |
|  "Int" parameter was coded on the execution statement.                        |
|                                                                               |
|  There are 4 value sets for int.                                              |
|                                                                               |
|  There are 1 values in set #1                                                 |
|  Low value of value #1 is 1                                                   |
|                                                                               |
|  There are 1 values in set #2                                                 |
|  Low value of value #1 is 2                                                   |
|                                                                               |
|  There are 1 values in set #3                                                 |
|  Low value of value #1 is 3                                                   |
|     and the high value is 5                                                   |
|                                                                               |
|                                                                               |
|  There are 3 values in set #4                                                 |
|  Low value of value #1 is 4                                                   |
|  Low value of value #2 is 5                                                   |
|  Low value of value #3 is 6                                                   |
|     and the high value is 9  <---------------------------------- End program output
|  CI Delete_file_connection standard_file=$echo file=output  <-- SCL command DELFC
|                                                                               |
+-------------------------------------------------------------------------------+
```

Figure 14-8. Execution Output of Procedure PTESTI

## Input/Output Differences

Input/output differences for NOS FORTRAN 5 and NOS/VE FORTRAN are described in topics as follows:

Buffer I/O

ENCODE/DECODE

OPENMS/READMS/WRITMS

O and Z Editing

Maximum Length of Formatted Records

Editing

Files INPUT and OUTPUT

### Buffer I/O

Some uses of buffer input/output, such as the unused bits value returned by LENGTH/LENGTHX and the size of the storage area to receive incoming data, are dependent on the number of characters per word. The parity indicator (p parameter) is ignored by NOS/VE FORTRAN, but must be present. The BUFFER statements are included in NOS/VE FORTRAN for compatibility only. Because buffers are not used in NOS/VE FORTRAN in the same way as in FORTRAN 5, BUFFER statements are generally not advantageous; unformatted READ and WRITE statements should be used instead.

The following example represents FORTRAN code on NOS that typically needs conversion. However, if the purpose is to transfer 1000 words of data on either NOS or NOS/VE, it should work on both.

```
DIMENSION ARRAY (1000)
  .
  .
  .
BUFFER OUT (8, 1) (ARRAY(1), ARRAY(1000))
I=UNIT(8)       |          |           |
   |            |          |           |
   |            |          |           The size of this record is 1000 words.  In NOS, this is 10000 6-bit
   |            |          |           bytes; in NOS/VE, 8000 8-bit bytes.
   |            |          |
   |            |          Used for parity on NOS; NOS/VE ignores it but requires its presence.
   |            |
   |            The unit must be cleared for subsequent I/O.
```

### ENCODE/DECODE

Most uses of ENCODE/DECODE involve packing and unpacking of characters within a word and are dependent on the number of characters per word. All usages should be checked. Conversion of ENCODE/DECODE to FORTRAN standard internal READ and WRITE is recommended.

Example of statements requiring conversion:

```
20  FORMAT (I3, F3.0, I4)
    ENCODE (10, 20, IVAR) INT1,REAL1,INT2
            |   |   |
            |   |   Boolean variable holds only 8 characters on NOS/VE (10 on NOS).
            |   |
            |   Format 20 specifies a 10-character transfer.
            |
            10 characters are to be transferred to IVAR.
            In NOS/VE, IVAR can hold only 8 characters.
```

### OPENMS/READMS/WRITMS

Most uses of OPENMS/READMS/WRITMS depend on a 10 character word length and on the record key being one word in length. Conversion to standard FORTRAN direct access I/O is recommended. The following example indicates conversion changes required:

```
CALL READMS(U, FWA, NWORDS, RECKEY)
                      |       |
                      |       |
                      |       If record key RECKEY is one word, its length changes from 10 on
                      |       NOS to 8 on NOS/VE.
                      |
                      Record size in 60-bit words on NOS and 64-bit words on NOS/VE.
```

### O and Z Editing

Under FORTRAN 5, reading a blank field with the Ow or Zw descriptor gives a minus zero. Under NOS/VE FORTRAN, no minus zero exists. (A positive zero is stored.) When it is necessary to see the value of a word on NOS/VE, use Z16 instead of O20 used on NOS.

All list items that contain boolean values should be declared type BOOLEAN.

### Maximum Length of Formatted Records

The maximum length of formatted records is reduced from 131,071 octal under FORTRAN 5 to 65,535 octal under NOS/VE FORTRAN.

### Editing

Hollerith data should be changed to character data where possible. Only eight characters fit into a Hollerith variable or constant on NOS/VE (not ten as on NOS).

All list items that contain boolean values should be declared type BOOLEAN.

### Files INPUT and OUTPUT

The system files INPUT and OUTPUT have been changed to NOS/VE standard files $INPUT and $OUTPUT.

FORTRAN programs that specify INPUT or OUTPUT on the PROGRAM statement will work. Programs that specify INPUT or OUTPUT on an OPEN statement might not work; change the specification on the OPEN statement to $INPUT or $OUTPUT.

Since $OUTPUT cannot be written to in a batch environment, you must connect $OUTPUT to a physical file containing data. You can use the PROGRAM statement or the SCL CREATE_FILE_CONNECTION command.

In a batch job, $INPUT is an empty file unless you connect $INPUT to a real file. FORTRAN recognizes EOP boundaries on files connected to $INPUT.

For information about standard files, see the file connection discussion in chapter 4, Common NOS/VE Commands.

NOS/VE FORTRAN provides a type of input/output called segment access files. Segment access files provide a fast and efficient way of sharing large blocks of data among FORTRAN programs. Segment access files are associated with common blocks, and are accessed through assignment statements rather than through input/output statements.

## Loader Differences

In general, the NOS CYBER Loader interfaces for use with FORTRAN 5 programs are used to manage field length to allow large programs to fit into central memory. With virtual memory on NOS/VE, this kind of memory management is not necessary. Therefore, you should remove or modify Loader interface code to migrate FORTRAN programs to NOS/VE.

A quick check for most code dependent on the CYBER Loader can be performed by using the ANSI parameter of the FTN5 command. For information about using this parameter to help migrate programs, see chapter 13.

The following pages briefly point out CYBER loader-dependent code.

### Overlays and OVCAPs

Overlays and OVCAPs are not meaningful in the NOS/VE environment and are not supported. You need to change the source code from a multi-main program structure to a one-main-program and related subroutines structure. Communication can occur via COMMON blocks; however, you need to eliminate duplicate common block and subroutine names.

All OVERLAY and OVCAP directives should be removed from programs being converted. PROGRAM statements in primary and secondary overlays should be changed to SUBROUTINE statements. The calls to OVERLAY, LOVCAP, XOVCAP, and UOVCAP are provided by NOS/VE FORTRAN but perform no operation.

### Static Memory Management

The static memory management routines are not supported by NOS/VE.

### Segment Loading

Segment loading is not supported under NOS/VE. To avoid conflicts in common block storage within segmented programs, you must change the names of nonglobal common blocks having the same name in parallel parts of the tree structure to be unique.

### Extensible Common Blocks

NOS/VE FORTRAN provides a feature called extensible common blocks. By using extensible common blocks, you can safely overindex arrays in common; the common block is automatically extended to avoid conflicts with other common blocks or data areas. Use of extensible common blocks is optional and is activated by the C$ EXTEND loader directive.

## FORTRAN Command

The FORTRAN command for NOS/VE FORTRAN differs from the FTN5 statement for FORTRAN 5. Parameter names have changed, new parameters are available, and certain FTN5 parameters are no longer supported.

The FORTRAN command follows the rules for all System Command Language (SCL) commands. If the FORTRAN command requires several physical lines, two or more periods at the end of a physical line indicate continuation to the following line. You can place the FORTRAN command in a line with other commands. Separate commands on a single line by semicolons. For example:

```
FORTRAN INPUT=$USER.INFIL BINARY_OBJECT=$USER.BIN ..
  LIST=LISTFIL OPTIMIZATION_LEVEL=HIGH
```

This command specifies permanent files INFIL and BIN as the source and binary files, respectively. The second line specifies HIGH optimization and that temporary file LISTFIL is to receive the output listing.

You can enter commands in interactive mode (through a terminal) or in batch mode. You can specify parameters on the command in any order. Separate parameters by a comma, or by one or more spaces.

The parameters of the NOS/VE FORTRAN command are briefly summarized in alphabetical order. The information about each command includes: the full parameter name, any parameter abbreviations, default values, and other allowed values. (Further information about the FORTRAN command appears in the FORTRAN Language Definition Usage manual.)

### BINARY_OBJECT or B

The BINARY_OBJECT parameter specifies the local file to receive the binary object code produced by the compiler. Options are:

| | |
|---|---|
| default | B=$LOCAL.LGO |
| B=file | Write object code to the specified file. |
| B=$NULL | Do not write object code to a file. |

### COMPILATION_DIRECTIVES or CD

The COMPILATION_DIRECTIVES parameter controls the recognition of C$ directives within the source program. Options are:

| | |
|---|---|
| default | CD=ON |
| CD=ON | Recognize C$ directives. |
| CD=OFF | Treat C$ directives as comments. |

### DEBUG_AIDS or DA

The DEBUG_AIDS parameter selects debugging options. Options are:

| | |
|---|---|
| default | DA=NONE |
| DA=PC | Produce information used by the loader to detect mismatches between actual and dummy arguments. |
| DA=DT | Produce line number tables, symbol tables, and source map loader tables. |
| DA=NONE | Do not produce line number tables, symbol tables, source map loader tables, and information for load-time argument checking. |
| DA=ALL | Selects the PC and DT options. |

### DEFAULT_COLLATION or DC

The DEFAULT_COLLATION parameter specifies the weight table to be used for evaluation of the character relational expressions by the CHAR and ICHAR functions. The options are:

| | |
|---|---|
| default | DC=FIXED |
| DC=F | (or DC=FIXED) Use fixed (7-bit ASCII) collating sequence. |
| DC=U | (or DC=USER) Use user-specified collating sequence. |

### ERROR or E

The ERROR parameter specifies the name of the file to receive compiler-generated error information. In the event of an error of ERROR_LEVEL-specified severity or higher, a diagnostic is written to the file specified by the ERROR parameter. If a listing file (LIST parameter) is also specified, the diagnostic is written to both files. Options are:

| | |
|---|---|
| default | Write error messages to standard file $ERRORS (which is automatically connected to file OUTPUT). |
| E=file | Write error messages to the specified file. |

### ERROR_LEVEL or EL

The ERROR_LEVEL parameter determines the severity level of errors to be printed on the output listing. Selection of a particular option specifies that level and all higher (more severe) levels. Options are (in order of increasing severity):

| | |
|---|---|
| default | EL=W |
| EL=T or EL=I | List trivial (informational), warning, fatal, and catastrophic errors. |
| EL=W | List warning, fatal, and catastrophic errors. |
| EL=F | List fatal and catastrophic errors. |
| EL=C | List catastrophic errors only. |

### EXPRESSION_EVALUATION or EE

The EXPRESSION_EVALUATION parameter controls the way the compiler evaluates expressions to perform optimizations. You can select multiple options among the last five. Options are:

| | |
|---|---|
| default | EE=NONE |
| EE=NONE | Perform normal optimizations (no options selected). |
| EE=C | Evaluate expressions according to precedence rules. |
| EE=ME | Do not perform optimizations which eliminate instructions that might cause execution errors. |
| EE=MP | Do not perform optimizations that change floating-point operations to a mathematically, but not computationally, equivalent form. |
| EE=OSM | Guarantees valid character assignment in character assignment statements of the form:<br><br>v=exp<br><br>where the character positions being defined in v are referenced in exp. |
| EE=R | Call intrinsic functions by reference rather than by value. |

### FORCED_SAVE or FS

The FORCED_SAVE parameter specifies whether or not the values of variables and arrays in the subprograms are to be retained after execution or a RETURN or END statement. Options are:

| | |
|---|---|
| default | FS=OFF |
| FS=ON | Save variable and array values after RETURN or END. |
| FS=OFF | Do not save variable and array values after RETURN or END. |

### INPUT or I

The INPUT parameter specifies the name of the file that contains the input source code. Options are:

| | |
|---|---|
| default | I=$INPUT, which automatically defaults to file INPUT. |
| I=file | Input source code is on the specified file. |

### LIST or L

The LIST parameter specifies the file to receive the compiler output listing. Options are:

| | |
|---|---|
| default | L=$LIST, which is automatically connected to standard file $NULL for interactive jobs and to file OUTPUT for batch jobs. |
| L=file | Write compiler output listing to the specified file. |

### LIST_OPTIONS or LO

The LIST_OPTIONS parameter specifies the information that is to appear on the compiler output listing. The information is written to the file specified by the LIST parameter. You can specify multiple options among the last six. For example:

```
LO=(A S R).
```

Options are:

| | |
|---|---|
| default | LO=S. |
| LO=NONE | Do not produce an output listing. |
| LO=A | List attributes of symbolic names. |
| LO=M | Produce symbol attribute list (same as A option), DO loop map, and common block map. |
| LO=O | Produce object code listing. |
| LO=R | Produce cross reference listing. |
| LO=S | List program source statements. |
| LO=SA | Same as S, but include lines turned off by C$ LIST |

### MACHINE_DEPENDENT or MD

The MACHINE_DEPENDENT parameter specifies whether the use of machine dependent capabilities within the program are to be diagnosed and if so, how severely. These capabilities include coding that depends on the number of characters in a word, such as use of the boolean data type, ENCODE and DECODE statements, and certain uses of BUFFER IN and BUFFER OUT. Options are:

| | |
|---|---|
| default | MD=NONE |
| MD=NONE | Do not diagnose machine-dependent usages. |
| MD=T or MD=I | Diagnose machine-dependent usages as trivial (informational) errors. |
| MD=W | Diagnose machine-dependent usages as warning errors. |
| MD=F | Diagnose machine-dependent usages as fatal errors. |

### ONE_TRIP_DO or OTD

The ONE_TRIP_DO parameter determines the manner in which DO loops are to be interpreted by the compiler. Options are:

| | |
|---|---|
| default | OTD=OFF |
| OTD=ON | Minimum trip count for DO loops is one. |
| OTD=OFF | Minimum trip count for DO loops is zero. |

### OPTIMIZATION_LEVEL or OL or OPTIMIZATION or OPT

The OPTIMIZATION_LEVEL parameter selects the level of optimization performed by the compiler. Options are:

| | |
|---|---|
| default | OL=LOW |
| OL=DEBUG | Generate object code modified for debugging. |
| OL=LOW | Perform minimum optimization. |
| OL=HIGH | Perform maximum optimization. |

### RUNTIME_CHECKS or RC

The RUNTIME_CHECKS parameter selects run-time range checking of subscripts and substrings. Options are:

| | |
|---|---|
| default | RC=NONE |
| RC=NONE | Do not perform run-time range checking. |
| RC=R | Perform range checking for character substring expressions. |
| RC=S | Perform range checking for subscript expressions. |
| RC=ALL | Select both R and S options. |

### SEQUENCED_LINES or SL

The SEQUENCED_LINES parameter specifies the sequencing format of the input source program. (Note that the FORTRAN sequenced format is not the same as the line-numbered format produced by NOS/VE. Line numbered source programs must not be written in sequenced format.) Options are:

| | |
|---|---|
| default | SL=OFF |
| SL=ON | Source program is in sequenced format. |
| SL=OFF | Source program is in nonsequenced format. |

### STANDARDS_DIAGNOSTICS or SD

The STANDARDS_DIAGNOSTICS parameter specifies whether the use of non-ANSI source statements are to be diagnosed and if so, how severely. Options are:

| | |
|---|---|
| default | SD=NONE |
| SD=NONE | Do not diagnose non-ANSI usages. |
| SD=T or SD=I | Treat non-ANSI usages as trivial (informational) errors. |
| SD=W | Treat non-ANSI usages as warning errors. |
| SD=F | Treat non-ANSI usages as fatal errors. |

### STATUS

The STATUS parameter defines an SCL status variable to be set by the compiler to contain information about errors that occur during compilation. The severity level of errors for which information is to be returned is determined by the TERMINATION_ERROR_LEVEL parameter. For more information about a status variable, see the discussion of the STATUS parameter in chapter 4, Common NOS/VE Commands.

| | |
|---|---|
| default | Do not return error status. |
| STATUS=var | Write error status code to the specified SCL variable. |

### TARGET_MAINFRAME or TM

The TARGET_MAINFRAME parameter is not supported at this release level. If you specify a value for the TARGET_MAINFRAME parameter, the value is ignored. You must indicate the position of the TARGET_MAINFRAME parameter with a comma if you specify the TERMINATION_ERROR_LEVEL or STATUS parameter by position.

### TERMINATION_ERROR_LEVEL or TEL

The TERMINATION_ERROR_LEVEL (TEL) parameter specifies the minimum error severity level for which the compiler is to return abnormal status. Information about errors having the specified or higher severity is returned. Options are:

| | |
|---|---|
| default | TEL=F |
| TEL=T or TEL=I | Return abnormal status for trivial (informational), warning, fatal, and catastrophic errors. |
| TEL=W | Return abnormal status for warning, fatal, and catastrophic errors. |
| TEL=F | Return abnormal status for fatal and catastrophic errors. |
| TEL=C | Return abnormal status for catastrophic errors only. |

## Other FORTRAN Feature Differences

Other feature differences between NOS FORTRAN 5 and NOS/VE FORTRAN are discussed in topics as follows:

7-Bit ASCII Code Set

Boolean Data Type

Default Collating Sequence

Double Precision Functions Referenced as Single Precision

Floating-Point Arithmetic

LEVEL Statement

Procedure Communication

PROGRAM Statement

SAVE Statement

### 7-Bit ASCII Code Set

For NOS the native code set is 6-bit display code; for NOS/VE, the native code set is 7-bit ASCII. This means that FORTRAN assumes that character data in files are in 6-bit display code on NOS and in 7-bit ASCII code on NOS/VE. Data manipulations can have different results because of the differences in the code sets. A summary of the major differences in the code sets is shown in table 14-3.

Table 14-3. Summary of 7-Bit ASCII and 6-Bit Display Code Differences

| Code Characteristic | 6-Bit Display Code | 7-Bit ASCII |
|---|---|---|
| Uppercase letters | Yes | Yes |
| Lowercase letters | Does not recognize case | Yes |
| Number of characters | 64 or 63 (site dependent) | 256 |
| Sequence of alphabetic vs numeric characters | Numeric first | Alphabetic first |
| Lowest value character | : (value 0 for 64-graphic set)<br>A (value 01 for 63-graphic set) | Space (value 32) |

The difference in native code sets makes applications on NOS/VE sensitive to uppercase and lowercase letters when they were not on NOS. For example, an application coded to terminate when "END" is entered terminates under NOS with "end". NOS/VE requires a match of uppercase and lowercase unless you code to allow other matches.

You can make an application insensitive to the case of letters by specifying a program collating sequence of ASCII6, which selects the OSV$ASCII6_FOLDED collating sequence (listed in appendix C). With ASCII6 specified, equivalent uppercase and lowercase letters have the same value.

To specify ASCII6, the program must include the CALL COLSEQ statement, and the FORTRAN command must specify DEFAULT_COLLATION=USER. These are shown below:

```
CALL COLSEQ ('ASCII6')
/fortran input=sourc_prog default_collation=user
```

All files containing character data on NOS/VE are assumed to use the 7-bit ASCII code set. Their collating sequence is assumed to be equivalent to the 7-bit ASCII sequence (that is, the weights of the characters are the same as the ASCII code set sequence for the characters). If the files used by the applications that you are migrating are dependent on other collating sequences, you need to ensure that the files are created on NOS/VE with the appropriate collating sequence specified. (This applies only to indexed sequential files.)

### Boolean Data Type

The boolean data types and operations (SHIFT, MASK, and so forth) are provided specifically for machine-dependent uses. Most uses will require program modification. For example, the following FORTRAN 5 example depends on a word length of 60 bits:

```
BOOL1 = SHIFT(BOOL2,42).OR.(BOOL3.AND.O"777777")
                        |          |
                        |          Builds a mask using octal constants in an 18-bit field.
                        |
                        +----------+--Boolean function for NOS/VE; not a logical operator.
                                      (Boolean function returns a value for a word; a logical
                                      operator returns a value for a bit.)
        |          |
        Boolean shift function and corresponding bit count (42 bits).
```

Declare boolean variables as type BOOLEAN. Change octal uses to hexadecimal.

On NOS, ones complement arithmetic was used. On NOS/VE, twos complement arithmetic is used. This means that masks or boolean values formed with negation will be different on NOS/VE.

### Default Collating Sequence

The default collating sequence established when the DEFAULT_COLLATION parameter is omitted from the FORTRAN command has been changed from USER to FIXED. For further discussion, see the preceding discussion in this chapter of the 7-Bit ASCII Code Set.

### Division Operation

Dividing by zero in NOS/VE causes a divide fault that terminates the program with an immediate fatal runtime error. In NOS, such a division causes a bad quotient that generates a run-time error when used as an operand in an expression.

### Double Precision Functions Referenced as Single Precision

Referencing double precision functions as single precision under FORTRAN 5 depends on register conventions that are not compatible with NOS/VE FORTRAN. All such uses should be removed. (These uses are not legal under FORTRAN 5, but some work.)

### Floating-Point Arithmetic

Differences in NOS and NOS/VE unrounded floating-point arithmetic can lead to different results if the source algorithm is numerically unstable. There are statistical methods available for a stable algorithm. Rounded floating-point arithmetic is not available on NOS/VE. Also, avoid .EQ. and .NE. comparisons to floating-point zero.

In NOS, a number that becomes too small due to exponent underflow is rounded to zero, and processing continues. In NOS/VE you can set an exponent underflow option with an SCL command. The default setting of the option is on, which means that a too small number results in processing being terminated with an immediate fatal runtime error. If you set the exponent underflow option off, NOS/VE treats exponent underflow the same way as NOS does.

### Hollerith Constants

Under NOS/VE FORTRAN, Hollerith constants are replaced by Boolean string constants, which are limited to 8 characters. Constants of the form, nHs, L"s", R"s", or "s", that exceed 8 characters are called extended Hollerith constants. You can use them only to pass actual arguments to external procedures.

### Hollerith Descriptors

Replace Hollerith descriptors with character descriptors.

### LEVEL Statement

The multilevel memory structure is not meaningful in the NOS/VE environment and is not supported. The LEVEL statement is accepted by the FORTRAN compiler but performs no operation.

### Procedure Communication

Any method of procedure communication, other than through common or an argument list, should be changed to use either common or an argument list.

### PROGRAM Statement

The file buffer length specifier on the NOS/VE FORTRAN PROGRAM statement is included for compatibility with FORTRAN 5, but is disregarded by the compiler. Because of the way in which buffers are used in the NOS/VE environment, assigning buffer lengths is not meaningful.

### SAVE Statement

Under NOS/VE FORTRAN, local variables and arrays in subprograms do not retain their values after an exit from the subprogram, unless the subprogram contains a SAVE statement or the FORCED_SAVE option is specified on the FORTRAN command.

# Migrating COBOL Programs 15

# Migrating COBOL Programs 15

Differences exist between NOS/VE COBOL and NOS COBOL 5.3. This chapter documents all major differences between these two versions of COBOL. Some minor differences could occur because of differences in the operating systems and in the character size (6 bits on NOS versus 8 bits on NOS/VE) used internally.

If only the syntax is different, the compiler automatically converts the old form of a statement to the new form in most cases. Specifying BASE_LANGUAGE=COBOL5 in the NOS/VE COBOL command converts some COBOL 5 syntax to NOS/VE COBOL syntax. Other differences require that you manually change the COBOL 5 source program before NOS/VE COBOL compilation and execution.

The differences between NOS/VE COBOL and COBOL 5 are divided into six major groups, which are subdivided into more specific topics. The major groups are:

Differences in statements, clauses, and sections

Differences relating to character and integer data

Differences relating to files

Differences in the COBOL compiler call

Differences for facilities, interfaces, and routines

Other differences

## Differences in Statements, Clauses, and Sections

The differences between NOS/VE COBOL and COBOL 5.3 for the following statements, clauses, and sections are discussed:

ACCEPT statement
ALTERNATE RECORD KEY clause
BLOCK CONTAINS clause
BLOCK COUNT clause
CALL statement
COPY statement
ENTER statement
INSPECT statement
READ statement
RECORD CONTAINS clause
RECORDING MODE clause
REDEFINES clause
REPLACE statement
RERUN statement
RESERVE AREAS clause
REWRITE statement
SECONDARY-STORAGE section
SET CODE-SET clause
SET PROGRAM COLLATING...SEQUENCE clause
SORT statement
SYNCHRONIZED clause
USE clause
USE FOR DEBUGGING declarative statement
USE FOR HASHING declarative statement
VALUE clause

### ACCEPT Statement

The default sizes of the various files on which data can be written have changed. However, the new default sizes should not cause any compatibility problems.

## ALTERNATE RECORD KEY Clause

The OMIT WHEN and USE WHEN phrases are not supported in NOS/VE COBOL. These are nonstandard phrases that are infrequently used, and few, if any, programs should be affected. Use the CREATE_ALTERNATE_INDEX utility, as described in the SCL Advanced File Management Usage manual, to create alternate record keys.

## BLOCK CONTAINS Clause

All programs must be converted to blocking appropriate for the files used on NOS/VE.

## BLOCK COUNT Clause

The BLOCK COUNT clause has been deleted. In NOS/VE COBOL, the home block count must be set by the INITIAL_HOME_BLOCK_COUNT parameter on the SET_FILE_ATTRIBUTES command.

## CALL Statement

NOS/VE COBOL allows a program-name up to 30 characters in length to be a program entry point. COBOL 5 allows only the first seven characters of a program name to become the program entry point.

If the COBOL 5 program relied on truncation to seven characters, the program no longer executes properly. If the program-name equivalence in the Fast Dynamic Loader (FDL) file were used in COBOL 5 and the coding took advantage of this, the program no longer works. Any programs that use proper names (that is, the name in the CALL statement is the same as the PROGRAM-ID of the called program) function correctly.

Any COBOL 5 program that calls a subprogram whose program-name specified in the PROGRAM-ID paragraph is greater than seven characters must have the name changed in the CALL statement to include the complete name of the called program.

Dynamic loading of programs is controlled by program management directives when programs are loaded or libraries are created. This requires manual conversion from the FDL file to these directives and an analysis of how to structure the program environment.

## COPY Statement

A NOS/VE COBOL source library is a Source Code Utility (SCU) source program file rather than an Update random program library. All embedded directives are now processed. The listing format is different for the structure of lines that are copied and in which replacement takes place.

Library conversion of the Update library source file is performed by SCU. See the SCU conversion commands in chapter 11 for more information.

The COPY statement differs slightly in processing in NOS/VE COBOL; it is not recognized in comment lines or comment entries. The COPY statement must observe the following rules:

A COPY statement must end with a period.

Only one COPY statement is allowed in each COBOL sentence.

A deck being copied by a COPY statement cannot contain a COPY statement.

## ENTER Statement

Language-name has been removed from the ENTER statement because all calls to other languages use a standard calling sequence. To convert ENTER statements in COBOL 5 programs, delete language-name. For compatibility, the words FTN5, FORTRAN-X, or COMPASS are ignored.

Procedure-name in the parameter list has been eliminated.

## INSPECT Statement

The BEFORE/AFTER option of the INSPECT...TALLYING...REPLACING statement (COBOL 5 Format 3) has been deleted for NOS/VE COBOL. Conversion can be accomplished by using separate INSPECT statements if AFTER was being used.

## READ Statement

In COBOL 5, if an unsuccessful START statement was followed by a READ NEXT statement, the contents of the record area were undefined. In NOS/VE COBOL, an error status is returned.

## RECORD CONTAINS Clause

Because of the differences in the numeric data types as indicated under Computational Data Types, the size of the record described for NOS/VE COBOL could be different from the size of the record described for COBOL 5. Manual conversion is necessary for each occurrence of this clause to ensure that the proper length is specified. (For further information on numeric data types, see the discussion Numeric Data Items.)

## RECORDING MODE Clause

Except for syntax checking, the RECORDING MODE clause is ignored in a NOS/VE COBOL program. For any COBOL 5 program that used this clause to set the parity on 7-track tapes, manual conversion is necessary to convert to a NOS/VE COBOL program. Parity must be set by a SET_FILE_ATTRIBUTE command.

## REDEFINES Clause

Any redefinition of records by the REDEFINES clause or automatically by the compiler could cause different results when executed under NOS/VE COBOL. Every USAGE IS COMP-n and SYNCHRONIZED clause must be checked to ensure that the result of execution of that clause is appropriate under NOS/VE COBOL and must be manually converted when necessary.

The runtime error INVALID BDP DATA can occur as a result of incorrect redefinition of numeric data. The error is issued by the operating system.

## REPLACE Statement

The replaced text shows optionally on the listing. Since it did not show at all in COBOL 5, this change should not cause any impact.

## RERUN Clause

Except for syntax checking, the RERUN clause is ignored in NOS/VE COBOL at this time. No checkpoint/restart facility exists in the NOS/VE operating system.

## RESERVE AREAS Clause

Except for syntax checking, the RESERVE AREAS clause is ignored in NOS/VE COBOL; therefore, conversion is not necessary. A NOS/VE COBOL program cannot specify buffer size because the operating system automatically assigns the appropriate buffer size.

## REWRITE Statement

In NOS/VE COBOL, the REWRITE statement is not allowed at this time for sequential files. No conversion is possible.

## Secondary-Storage Section

A facility to specify secondary storage is not needed with the NOS/VE operating system. Virtual memory removes the need for large external storage for arrays. Conversion of a COBOL 5 program using secondary storage is necessary. Remove the Secondary-Storage Section header and include under the Working-Storage Section header all items that were in the Secondary-Storage Section.

The Secondary-Storage Section header is ignored if the Working-Storage Section header exists; otherwise, the Working-Storage Section header is created. The data is considered to be part of the Working-Storage Section, and a trivial diagnostic is issued.

## SET CODE-SET Clause

The CODE-SET option of the SET statement is not supported in NOS/VE COBOL. The only CODE-SET supported with this statement by COBOL 5 is UNI. This is not supported as a CODE-SET in NOS/VE COBOL. No conversion is possible.

## SET PROGRAM COLLATING SEQUENCE Clause

In COBOL 5, the SET PROGRAM COLLATING SEQUENCE clause caused the first OPEN (OPEN OUTPUT) on an indexed sequential file to use the program collating sequence as the indexed sequential file collating sequence. In NOS/VE COBOL, the clause has no effect on the indexed sequential file, unless BASE_LANGUAGE=COBOL5 is specified on the COBOL command.

## SORT Statement

Under COBOL 5, the default was to sort duplicates (that is, records whose sort keys have the same value) in the sequence of arrival within each set of duplicates. For the same result in a NOS/VE COBOL program, specify the WITH DUPLICATES IN ORDER phrase in the SORT statement. This phrase was available under COBOL 5, and if specified, no conversion is necessary. Manual conversion is necessary if the phrase was not specified and the application requires sorting in this particular order. Also, if C.SORTP was called in COBOL 5, the keys would not be in order. Using the WITH DUPLICATES IN ORDER phrase results in additional overhead in the sort operation.

## SYNCHRONIZED Clause

The SYNCHRONIZED clause can cause different results in NOS/VE COBOL. Manual conversion is required for the SYNCHRONIZED clause. You must determine the appropriate conversion after analyzing the reason for using this clause in the COBOL 5 program and considering the creation of FILLER items.

Using the SYNCHRONIZED clause in COBOL 5 often improves the efficiency of the code generated. Using this clause in NOS/VE COBOL normally does not affect the efficiency of the code generated. Problems with the SYNCHRONIZED clause occur primarily with group operations, record areas, and the REDEFINES clause.

### USE Clause

The USE clause differs in NOS/VE COBOL and COBOL5. In NOS/VE COBOL, it is not used to set file attributes. You can set file attributes with the NOS/VE SET_FILE_ATTRIBUTES command.

### USE FOR DEBUGGING Declarative Statement

Debugging declaratives are not supported in NOS/VE COBOL. The WITH DEBUGGING MODE clause applies only to the compilation of debugging lines. Debugging declaratives are compiled as if the WITH DEBUGGING MODE clause had not been specified.

### USE FOR HASHING Declarative Statement

The USE FOR HASHING declarative statement is not provided by NOS/VE COBOL. Removing this statement should not affect the rest of the program. You can write your own hashing procedure using the CYBIL language. For more information, see the SCL Advanced File Management Usage manual.

### VALUE Clause

In COBOL 5 if no VALUE clause is specified for an item, an initial value of spaces is supplied. In NOS/VE COBOL if no VALUE clause is specified for an item, its contents are undefined. To convert your program so that the desired items are initialized when executing with NOS/VE, you can either add a VALUE clause, or use the INITIALIZE or MOVE statement.

## Differences Relating to Character and Integer Data

The differences relating to character and integer data are discussed in the following topics:

Arithmetic Expressions or Arithmetic Operations

Boolean Items

Character Set

Code Sets

Collating Sequence

Computational Data Types

Numeric Items

Signs

SIZE ERROR

### Arithmetic Expressions or Arithmetic Operations

Some arithmetic operations in NOS/VE COBOL, especially those involving division or exponentiation, can result in answers that differ from those answers given in COBOL 5. Most floating point operations give different answers. The quantities of these differences should be very slight (plus or minus 1 in the low order position).

Few, if any, programs should be affected by the differences in arithmetic operations.

## Boolean Items

Boolean data items and operators have been deleted. Complete recoding is necessary.

## Character Set

Uppercase and lowercase characters are allowed in NOS/VE COBOL. This could cause confusion because in COBOL 5 all lowercase characters are mapped into uppercase characters; however, no compatibility problem should occur.

The characters for minus zero are different. In COBOL 5, < and ! were plus and minus zero, respectively. In NOS/VE COBOL, these characters are accepted, but { and } are the preferred characters for plus and minus zero, respectively. COBOL 5 also accepts a number of other characters, which should not cause any problems.

## Code Sets

For NOS/VE COBOL, the system code set is the 7-bit ASCII code set. For COBOL 5, the system code set was an installation option and could be either the CDC 63 or 64-character 6-bit display code set.

For application programs using the CDC code set, you should specify the CDC code set as the program code set by using the ALPHABET clause in the SPECIAL-NAMES paragraph. See the NOS/VE COBOL Usage manual for an explanation of the ALPHABET clause in the SPECIAL-NAMES paragraph to determine the proper code set.

The UNIVAC fieldata automatic code translation is no longer supplied; however, the collating sequence is still available. If you specify an alphabet-name associated with UNI on a CODE-SET clause, an error message is generated.

An alphabet-name associated with EBCDIC was ignored in COBOL 5, but is processed in NOS/VE COBOL.

## Collating Sequence

The default collating sequence is ASCII, where the default might have been CDC-64 in COBOL 5. The COBOL 5 default depends on the installation parameter, IP.CSET. To convert the program, specify the CDC collating sequence in the SPECIAL-NAMES paragraph and either in a SET statement or in the OBJECT-COMPUTER paragraph referencing the proper alphabet-name.

The fact that there are now 256 characters instead of 64 could cause some problems with collating sequences. However, any program that uses only the 64 character subset should process correctly.

## Computational Data Types

NOS/VE COBOL computational data types differ from those in COBOL 5. You must manually convert redefinitions of and references to group items containing the USAGE clause that specifies COMP-n. Calls to other languages (such as FORTRAN) that contain COMP-n parameters give indeterminate results in a NOS/VE COBOL program; you must manually convert these calls to specify the correct parameters.

Using BASE_LANGUAGE=COBOL5 in the NOS/VE COBOL command automatically converts computational data types as shown in the subsequent list.

COBOL 5 computational data types must be converted as follows:

| COBOL 5 | NOS/VE COBOL |
|---|---|
| COMP | DISPLAY |
| COMP-1 | COMP |
| COMP-2 | COMP-1 |
| — | COMP-2 (new) |
| COMP-4 | COMP or COMP-3 |

### COMP (COBOL 5) to DISPLAY (NOS/VE COBOL)

The COBOL 5 computational data type COMP must be converted to the NOS/VE computational data type DISPLAY.

The internal representation of the COBOL 5 computational data type COMP is display code numeric.

The internal representation of the NOS/VE COBOL computational data type DISPLAY is display code numeric.

### COMP-1 (COBOL 5) to COMP (NOS/VE COBOL)

The COBOL 5 computational data type COMP-1 must be converted to the NOS/VE data type COMP.

The internal representation of the COBOL 5 computational data type COMP-1 and the NOS/VE COBOL computational data type COMP are as follows:

| COMP-1 (COBOL 5) | COMP or BINARY (NOS/VE COBOL) |
|---|---|
| Numeric class | Numeric class |
| Occupies full computer word of 60 bits | Decimal numeric value |
| Stored as a 48-bit binary integer, right-aligned | Stored as a binary integer |
| | Size depends on PIC clause |
| FORTRAN integer | FORTRAN integer type (if size equals 18 and synchronized) |

### COMP-2 (COBOL 5) to COMP-1 (NOS/VE COBOL)

The COBOL 5 computational data type COMP-2 must be converted to the NOS/VE COBOL computational data type COMP-1.

The internal representation of the COBOL 5 computational data type COMP-2 and the NOS/VE computational data type COMP-1 are as follows:

| COMP-2 (COBOL 5) | COMP-1 (NOS/VE COBOL) |
|---|---|
| Single precision floating point number | Single precision floating point number |
| Occupies one word of 60 bits | Occupies one word of 64 bits |
| No PICTURE clause | No PICTURE clause |
| Value is a signed normalized floating point number | Value is a signed normalized floating point number |
| Up to 14 significant digits | Up to 14 significant digits |
| FORTRAN real type | FORTRAN real type |

### COMP-2 (NOS/VE COBOL)

The NOS/VE COBOL computational data type COMP-2 has no COBOL 5 equivalent.

The internal representation of the NOS/VE COBOL computational data type COMP-2 is as follows:

- Double precision floating point number
- Up to 28 significant digits

### COMP-4 (COBOL 5) to COMP or COMP-3 (NOS/VE COBOL)

The COBOL 5 computational data type COMP-4 must be converted to the NOS/VE COBOL computational data types COMP or COMP-3.

The internal representations of the COBOL 5 computational data type COMP-4 and the NOS/VE computational data type COMP-3 are as follows (for an explanation of NOS/VE COBOL computational data type COMP, see the preceding discussion on COBOL 5 COMP-1):

| COMP-4 (COBOL 5) | COMP-3 (or PACKED-DECIMAL) (NOS/VE COBOL) |
|---|---|
| Numeric class | Numeric class |
| Size depends on PIC clause | String of 4-bit representations of numeric digits packed two per byte with optional sign as rightmost 4 bits in rightmost byte |
| Binary Integer | |
| Maximum of 48 bits | |
| Signed or unsigned | |

### Numeric Data Items

In NOS/VE COBOL, use of a numeric item with nonnumeric contents causes the job to be aborted by the hardware. This results in the operating system message INVALID BDP DATA.

### Signs

The NOS/VE COBOL sign representation for numeric items can differ from the COBOL 5 sign representation. If an item has an overpunch sign (no SEPARATE clause in its Data Description entry), a sign could be generated for positive values. In COBOL 5, this normally would not happen.

### SIZE ERROR

With COBOL 5, a MODE control statement could be used to cause execution of the SIZE ERROR statements when floating point overflow occurred. In NOS/VE there are two ways to cause execution of the ON SIZE ERROR statements when floating point overflow occurs:

- The preferred method is to set the appropriate parameters on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, CHANGE_PROGRAM_DESCRIPTION, or EXECUTE_TASK commands. Information about these commands is in the SCL Object Code Management Usage manual.

- The less efficient method is to use RUNTIME_CHECKS=R in the COBOL command.

## Differences Relating to Files

The differences between NOS/VE COBOL and COBOL 5.3. relating to files are described in the following topics:

Actual-key file organization

Direct file organization (FO=DA)

File-names

File status

Indexed sequential file organization

Record types

Relative file organization

Word-address file organization

### Actual-Key File Organization

NOS/VE COBOL does not support the actual-key file organization. An actual-key file must be converted to an indexed sequential file.

To convert the file, change the organization to INDEXED, set the key to zero before opening the file, and add 1 to the key after each WRITE to the file.

Since the record numbers can differ in the indexed sequential file from those in the actual-key file, check the logic of the program to ensure that no problems arise.

### Direct File Organization

NOS/VE supports the direct access file organization; however, in NOS/VE COBOL, the BLOCK COUNT clause is not used to establish the initial number of home blocks. Instead, use the INITIAL_HOME_BLOCK_COUNT parameter on the SET_FILE_ATTRIBUTES command.

The USE FOR HASHING declarative is not provided by NOS/VE COBOL. You can write your own hashing procedure in the CYBIL language, and you can specify a user-defined hashing procedure with the HASHING_PROCEDURE_NAME attribute. For more information, see the SCL Advanced File Management Usage manual.

## File-Names

The rules for formation of local file-names in NOS/VE COBOL differ from those in COBOL 5. NOS/VE COBOL allows several special characters in the formation of names. The standard system files in NOS/VE COBOL require leading $ characters; this forces the file-names to be enclosed in quotation marks (which is non-ANSI) in NOS/VE COBOL programs. NOS/VE COBOL provides the special names SYSTEM-INPUT-FILE and SYSTEM-OUTPUT-FILE for audit or ANSI use.

Because almost all programs assign files to INPUT or OUTPUT, the name changes could have a large impact. If you use only defaults for $INPUT or $OUTPUT in commands (that is, you do not use CONNECT_FILE), INPUT and OUTPUT reference the same files as $INPUT and $OUTPUT.

Note that INPUT is a null file in batch mode. You cannot put any data in it, so change the program to use a different file and use the COLLECT_TEXT command to place data in that file. For information about COLLECT_TEXT, see chapter 4, Common NOS/VE Commands.

## File Status

The codes available in the file status data item have been expanded to correspond to the ANSI proposed standard. The only impact is that codes that resulted in a 90 or 99 status and aborted the job no longer abort the job and now have another status code.

## Indexed Sequential File Organization

NOS/VE COBOL supports the indexed sequential file organization and alternate keys.

The collating sequence of an indexed sequential file is NATIVE rather than that of the program in which the file was created. This change is due to an ANSI interpretation. If BASE_LANGUAGE=COBOL5 is in the COBOL command, the program collating sequence is used. If you omit the AUDIT parameter in the COBOL command, you can override the default NATIVE collating sequence by specifying the collating sequence on a SET_FILE_ATTRIBUTES command prior to opening the file.

## Record Types

The record types specified for some files in NOS/VE COBOL can differ from those in COBOL 5. Most files must be converted for use with NOS/VE COBOL; therefore, a manual conversion of the program is necessary to specify the files being used with the NOS/VE operating system. Any files with unusual record types that were used with COBOL 5 programs require manual conversion for use with NOS/VE COBOL programs. If record type Z was used in COBOL 5, TRUNCATE_SPACES must be specified in the USE literal with NOS/VE COBOL. This is especially true if RT=Z was specified in a NOS FILE command.

## Relative File Organization

NOS/VE COBOL supports the relative file organization; however, the record structure is different. In NOS/VE COBOL, there is a one-byte trailer rather than a one-word header for each record. There is no longer a file header. If a relative file was being processed by some other language, that process must be changed.

### Word-Address File Organization

The word-address file organization (COBOL 5) has been changed to byte-address (also called byte addressable) file organization (NOS/VE COBOL). All references to WORD-ADDRESS must be changed to BYTE-ADDRESS.

Usually, if the records are written sequentially and the keys are saved, no conversion is necessary.

If conversion is necessary, consider the following items when converting the logic of the program:

In a word-address file, the first address (key) is 1; in a byte addressable file, the first address (key) is 0.

For a file used with a COBOL 5 program, a word contains 10 characters; for a file used with a NOS/VE COBOL program, a byte contains one character. The formula, COMPUTE B = (W - 1) * 10, can be used to convert a word-address file to a byte addressable file. In this formula, B is the byte-address and W is the word-address.

## Compiler Call

Some parameters available on the COBOL5 command are not available on the NOS/VE COBOL command. These deleted parameters include the following:

| Parameter | Meaning |
|---|---|
| ANSI=NOEDIT | Prevented the editing of numeric display items. |
| ANSI=77LEFT | Caused left synchronization of level 77 items. |
| BL | Caused the printing of a burstable listing. |
| CC1 | Converted COMP to COMP-1; not needed because COMP is now binary. |
| D | Identified a data base subschema. |
| FDL | Specified the file for Fast Dynamic Loader processing. |
| MSB | Indicated the main subprogram; any subprogram can now be the main subprogram. |
| PD | Specified the print density for printable output; now a file attribute (PRINT_DENSITY). |
| PS | Specified the number of lines per screen; now a file attribute (PAGE_SIZE). |
| PSQ | Specified the source of sequence numbers to be used for diagnostics; now a file attribute (LINE_NUMBER). |
| PW | Specified the number of characters per output line; now a file attribute (PAGE_WIDTH). |
| TAF | Executed the program through the Transaction Facility. |
| TDF | Specified the file for a termination dump. |
| U | Specified the file for generation of input to the UPDATE utility. |
| UC1 | Converted COMP-1 items to integer format. |

## NOS/VE COBOL Command

The COBOL command calls the COBOL compiler and selects various compiler options. The COBOL command follows the rules for all System Command Language (SCL) commands.

If several physical lines are required for the COBOL command, two or more periods at the end of a physical line indicate continuation to the following line. You can place the COBOL command in a line with other commands. Separate commands on a single line by semicolons. You can enter commands in interactive mode (through a terminal) or in batch mode.

You can specify parameters on the command in any order. Separate parameters by a comma, or one or more spaces. You can specify the INPUT, BINARY_OBJECT, and LIST parameters by position without the parameter name if you specify them respectively as the first, second, and third parameters.

### COBOL Command Format

The format of the command and the optional parameters are shown below. The three positional-dependent parameters are listed first. The remaining parameters are listed alphabetically.

```
COBOL
    INPUT=file
    BINARY_OBJECT=file
    LIST=file
    AUDIT=boolean
    BASE_LANGUAGE=keyword
    DEBUG_AIDS=list of keyword
    ERROR=file
    ERROR_LEVEL=keyword
    EXTERNAL_INPUT_FILE=file
    FED_INFO_PROCESSING_STANDARD=list of keyword
    INPUT_SOURCE_MAP=file
    LEADING_BLANK_ZERO=boolean
    LIST_OPTIONS=list of keyword
    LITERAL_CHARACTER=string
    OPTIMIZATION_LEVEL=keyword
    RUNTIME_CHECKS=list of keyword
    STANDARDS_DIAGNOSTICS=list of keyword
    STATUS=status variable
    SUBPROGRAM=boolean
```

## COBOL Command Parameters

The parameters of the COBOL command are described in the following paragraphs. The three positional-dependent parameters are listed first. The remaining parameters are listed alphabetically.

### INPUT or I

The INPUT parameter specifies the source input file for the COBOL compiler. Options are:

| | |
|---|---|
| Omitted | Uses the system file $INPUT to hold source input. |
| I=file | Uses the specified file to hold source input. |

### BINARY, BINARY_OBJECT, B, or BO

The BINARY parameter specifies the file reference to which binary output from compilation is written. Options are:

| | |
|---|---|
| Omitted | Binary object code is written to the file LGO. (Same as BINARY=LGO.) Default positioning is BOI. |
| B=$NULL | Binary object code output is not written. |
| B=file | Binary object code is written to the specified file. |

### LIST or L

The LIST parameter specifies the file to which the COBOL compiler writes the source listing and other readable output. Options are:

| | |
|---|---|
| Omitted | Writes source listing, diagnostics, and information selected by the LIST_OPTIONS parameter to file $LIST. |
| L=file | Writes source listing, diagnostics, and information selected by the LIST_OPTIONS parameter to the specified file. |
| L=$NULL | Does not write source listing, diagnostics, and information selected by the LIST_OPTIONS parameter to a file. |

### AUDIT, A, or AUD

The AUDIT parameter indicates whether the program is being run for Federal Software Testing Center (FSTC) audit testing. Selection of this option also selects the ERROR_LEVEL=I and STANDARDS_DIAGNOSTICS=(I,ANSI) parameters. When you select the AUDIT option, numeric items referenced in a DISPLAY statement produce unedited results. See the DISPLAY statement for details. Options are:

| | |
|---|---|
| Omitted | Does not select AUDIT option. |
| A=TRUE | Performs FSTC audit testing. |

When AUDIT=TRUE, non-ANSI reserved words are not recognized as reserved words and, therefore, can be user-defined words. CDC extension keywords are diagnosed as illegal.

If a group item containing a variable-occurrence data item is used as a receiving item, only that part of the table area specified by the value of the DEPENDING ON data item is affected.

### BASE_LANGUAGE or BL

The BASE_LANGUAGE (BL) parameter allows the user to compile programs with syntax based on different base languages. This is a single value parameter. Options are:

| | |
|---|---|
| Omitted | Same as BL=ANS85 |
| BL=ANS74 | Compiles programs whose syntax is based on the 1974 ANSI COBOL standard. |
| BL=ANS85 | Compiles programs whose syntax is based on the 1985 ANSI COBOL standard. |
| BL=COBOL5 | Compiles programs written for compilation by COBOL 5. |

### DEBUG_AIDS or DA

The DEBUG_AIDS (DA) parameter specifies debugging options. You can select multiple options, which are separated by either a space or a comma. Options are:

| | |
|---|---|
| Omitted | Same as DA=NONE. |
| DA=NONE | Does not select any of the debugging options applicable to this parameter. |
| DA=ALL | All of the available options are selected except SY and NONE. |
| DA=DS | Compiles debugging lines in the source program (lines with a D in character position 7). |
| DA=DT | Generate line number, symbol tables, and source map loader tables as part of the object code. |
| DA=OC | Continues producing object code from the source code and issues compilation-time diagnostics regardless of errors in the source code and regardless of the severity of any errors. Any line of code given an F diagnostic usually results in an abort at execution time with an appropriate diagnostic. If not specified, no object code is produced if any F or C errors are detected (see ERROR_LEVEL). |
| DA=SY | Performs only syntax checking; does not generate any executable code. You cannot select this option if you selected the OC option. Selecting the SY option significantly decreases compilation time. If you do not select the OC option and errors above the W level are detected, SY is automatically selected. |
| DA=TR | Produces flow tracing of all paragraphs executed. |

### ERROR or E

The ERROR parameter specifies the file to which error listing information is written. If the same file is specified by both the ERROR parameter and the LIST parameter, a diagnostic is written twice. Options are:

| | |
|---|---|
| Omitted | Writes error information specified by the ERROR_LEVEL parameter to the $ERRORS file. |
| E=file | Writes error information specified by the ERROR_LEVEL parameter to the specified file. |

### ERROR_LEVEL or EL

The ERROR_LEVEL parameter indicates the severity of the errors to be printed in the file specified by the ERROR parameter. The error levels in increasing order of severity are N, T, W, F, and C. Specifying a particular level selects that level and all of the more severe levels. You can specify a single value for this parameter. Options are:

| | |
|---|---|
| Omitted | Lists errors of severity levels W, F, and C. |
| EL=NONE | Does not list any errors. |
| EL=I | Lists trivial errors, plus all errors of levels W, F, and C.<br><br>Level I errors indicate suspicious usage, although the syntax is correct. EL=I is required to obtain a listing of the messages created by FIPS parameter.<br><br>See the STANDARDS_DIAGNOSTICS parameter discussion to choose a severity level for trivial errors. |

EL=T — Same as EL=I.

EL=W — Lists warning errors, plus all errors of levels F and C.

Level W errors indicate that the syntax of the statement is incorrect and the compiler has made an assumption and continued compilation.

EL=F — Lists fatal errors, plus all level C errors.

Level F errors prevent compilation of the statement. Unresolvable semantic errors and propagated errors caused by earlier level F errors are among the causes of level F errors. If you specify DEBUG_AIDS=OC, code is generated that usually causes an execution-time abort; otherwise, no code is generated.

EL=C — Lists catastrophic errors only.

Level C errors are fatal to compilation of the current program. Compilation resumes at the Identification Division header of any program immediately following without an intervening file boundary. Level C errors are caused by errors in the compiler, by input-output errors on compiler scratch files, or by other system errors.

### EXTERNAL_INPUT, EI, or EX_INPUT

The EXTERNAL_INPUT parameter specifies the Source Code Utility (SCU) library file to be used for COPY statements. Options are:

Omitted — Does not use SCU library file. (Same as EXTERNAL_INPUT=$NULL.)

EI=file — Specifies SCU library file.

### FED_INFO_PROCESSING_STANDARD or FIPS

The FED_INFO_PROCESSING_STANDARD parameter specifies diagnosing input source statements that do not conform to the standards in some part of the 1985 FEDERAL INFORMATION PROCESSING STANDARDS (FIPS) COBOL subset. You can specify which part of the 1985 FIPS COBOL subset; either the entire COBOL subset or some of its optional modules. The FIPS parameter also permits diagnosing syntax identified in the obsolete category of American National Standard Programming Language COBOL, X3.23-1985. The FIPS parameter has meaning only when BASE_LANGUAGE=ANS85. If BASE_LANGUAGE=ANS74 or BASE_LANGUAGE=COBOL5, this parameter is ignored.

The n that terminates several of the keywords specifies a level and can be only the integers 1 or 2. Specifying two different levels of the same keyword (such as CL1 and CL2) is an error. Specifying OMLn with CLn, DLn, RWLn, or SLn is an error.

When you specify this parameter, also specify the STANDARDS_DIAGNOSTICS parameter to set the severity level of any diagnostics issued. The allowed keywords follow:

Omitted — Same as FIPS=NONE.

FIPS=NONE — Does not select any option.

FIPS=CLn — Issues diagnostics for syntax that does not conform to level n of FIPS COBOL for the COMMUNICATIONS optional module.

FIPS=DLn — Issues diagnostics for syntax that does not conform to level n of FIPS COBOL for the DEBUG optional module.

FIPS=OBSOLETE or FIPS=O — Issues diagnostics for syntax identified in the obsolete category of the 1985 ANSI COBOL standard.

FIPS=OMLn — Issues diagnostics for syntax that does not conform to level n of FIPS COBOL for all optional modules.

FIPS=RWLn — Issues diagnostics for syntax that does not conform to level n of FIPS COBOL for the REPORT WRITER optional module.

FIPS=SLn — Issues diagnostics for syntax that does not conform to level n of FIPS COBOL for the SEGMENTATION optional module.

FIPS=SM — Issues diagnostics for syntax that does not conform to the MINIMUM subset for FIPS COBOL.

FIPS=SI — Issues diagnostics for syntax that does not conform to the INTERMEDIATE subset for FIPS COBOL.

FIPS=SH — Issues diagnostics for syntax that does not conform to the HIGH subset for FIPS COBOL.

### INPUT_SOURCE_MAP or ISM

The INPUT_SOURCE_MAP parameter specifies the name of the file that contains the source map describing the contents of the source input file. Options are:

Omitted — The input source map file is constructed during compilation based on the contents of the source input file.

ISM=file — File that contains the source map of the source input file. An example of a source map file is the OUTPUT_SOURCE_MAP file created by the EXPAND_DECK commands of the Source Code Utility (SCU).

### LEADING_BLANK_ZERO or LBZ

The LEADING_BLANK_ZERO parameter specifies that leading blanks in numeric fields are treated as zeros in arithmetic statements and comparisons. If you select LBZ, performance degrades severely. You should use this parameter only for checkout purposes until all data is converted to proper numeric format. Options are as follows:

Omitted — Specifies that numeric fields containing blanks are in error.

LBZ=TRUE — Treats all leading blanks in numeric fields as zeros in arithmetic statements and comparisons.

### LIST_OPTIONS or LO

The LIST_OPTIONS parameter specifies the options of extra information that are to appear in the file specified by the LIST parameter. You can specify multiple options. Options are:

Omitted — Lists source program. (Same as LIST_OPTIONS=S.

LO=NONE — Does not select any of the available options.

LO=M — Produces map of data-names and procedure-names, their physical storage, and a list of their attributes.

LO=O — Lists generated object code with instruction mnemonics.

LO=R — Produces cross-reference listing of all data-names and procedure-names. The listing shows the locations of definition and use within the program. Only items referenced in the program are listed.

LO=RA Produces cross-reference listing of all data-names and procedure-names, whether referenced within the program or not.

LO=S Lists source program.

LO=SA Lists all program source statements; the listing includes those lines turned off by a source embedded NOLIST directive.

### LITERAL_CHARACTER or LC

The LITERAL_CHARACTER parameter changes the character that delimits nonnumeric literals in program source code. This parameter is normally used to specify the single quotation mark or apostrophe. The value LC=NONE is not allowed. Options are:

Omitted Uses quotation mark to delimit nonnumeric literals.

LC=''''  Uses apostrophe or single quote to delimit nonnumeric literals.

LC='"'  Uses quotation mark to delimit nonnumeric literals.

### OPTIMIZATION_LEVEL, OL, OPTIMIZATION, or OPT

The OPTIMIZATION_LEVEL parameter selects the level of optimization performed by the compiler. Options are:

Omitted Same as OL=LOW.

OL=LOW Selects minimum level of optimization.

OL=DEBUG Selects minimum optimization with code modified for use by Debug. (Results in slower execution than OL=LOW.)

### RUNTIME_CHECKS or RC

The RUNTIME_CHECKS parameter selects execution-time checking of reference modifiers, subscripts, or index references. Options are as follows:

Omitted Same as RC=NONE

RC=NONE Does not select any options.

RC=R Checks reference modifiers to see if they fit within the subject data item. If an error is detected, the program aborts and an appropriate diagnostic is issued. Turns on system arithmetic overflow condition checking. If an arithmetic result is too large for the field which is to contain it, an appropriate system diagnostic is issued.

RC=S Checks all subscripts or index references for validity at execution time. If an error is detected, the program aborts and an appropriate diagnostic is issued.

RC=ALL Selects the R and S options.

### STANDARDS_DIAGNOSTICS or SD

The STANDARDS_DIAGNOSTICS parameter specifies diagnosing input source statements that do not conform to American National Standard Programming Language COBOL, X3.23-1985. Options are:

| | |
|---|---|
| Omitted | Same as SD=NONE. |
| SD=NONE | Does not select any option. |
| SD=(severity,ANSI) | Specifies that source statements not conforming to the 1985 American National Standard Programming COBOL are to be diagnosed. When you specify this option, also specify the ERROR_LEVEL parameter and value. |

The severity is one of the following:

- I — Non-standard usages result in informational diagnostics.
- W — Non-standard usages result in warning diagnostics.
- F — Non-standard usages result in fatal diagnostics.

### STATUS

The STATUS parameter specifies the name of the SCL status variable that is set by the compiler and contains information about error conditions occurring during compilation. The severity level of errors for which information is returned is determined by the ERROR_LEVEL parameter. Options are:

| | |
|---|---|
| Omitted | No command status variable is set. |
| STATUS=name | Sets status variable name. The status variable consists of three fields. After compilation, these fields contain information as follows: |

Normal field — Contains value of FALSE if compile-time error conditions occurred, and value of TRUE if no error conditions occurred. If the value is TRUE, then the remaining fields are undefined.

Condition field — Indicates the specific error condition that occurred.

Text field — A string that contains substitution values for the error message template normally displayed for the particular error.

### SUBPROGRAM or SP

The SUBPROGRAM parameter indicates that the source program is to be compiled as a subprogram instead of as a main program. Options are:

| | |
|---|---|
| Omitted | Compiles source program as a main program (same as SP=FALSE). |
| SP=TRUE | Compiles source program as a subprogram. |
| SP=FALSE | Compiles source program as a main program. |

## Differences for Facilities, Interfaces, and Routines

The differences between NOS/VE COBOL and COBOL 5.3 for facilities, interfaces, and routines are discussed as follows:

Communications facility (MCS)

CYBER Database Control System (CDCS) interface

FORTRAN interface

Paragraph trace facility

Segmentation facility

Termination dump facility

Utility routines

### Communication Facility (MCS)

The Communication facility is not available in NOS/VE COBOL. Programs specifying MCS can be compiled, but any attempt to execute MCS statements causes a program abort.

### CYBER Database Control System Interface

No interface to the CYBER Database Control System (CDCS) is available with the NOS/VE operating system.

### FORTRAN Interface

You must change integer parameters that are passed to FORTRAN subprograms in a NOS/VE COBOL program. The parameters passed should be described as follows to cause a full word alignment on a word boundary:

```
PICTURE IS S9(180)  USAGE IS COMP
    SYNCHRONIZED LEFT
```

You also should delete language-name (FORTRAN-X or FTN5) from the ENTER statement in a NOS/VE COBOL program.

You can use either a CALL statement or an ENTER statement to call a FORTRAN program. Use of a CALL statement is recommended.

Input-output is fairly compatible. The severe restrictions necessary in COBOL 5 are no longer necessary; however, problems could still arise if other than PRINT statements are used in FORTRAN.

### Paragraph Trace Facility

Records created by use of NOS/VE COBOL are 48 characters in length and are V type records; records created by use of COBOL 5 are 50 characters in length and are Z type records.

The calls required to activate the Paragraph Trace facility change from C.xxxTR to CBP$TRACE_ON or CBP$TRACE_STOP. Using BASE_LANGUAGE=COBOL5 in the NOS/VE COBOL compiler call causes the appropriate conversion to be performed on the names of the calls.

The name of the trace file has changed from COBTRFL to CBF$TRACE_FILE. No conversion is done.

## Segmentation Facility

NOS/VE COBOL does not support segmentation because there are no overlays on NOS/VE.

The concept of initial state overlays is not valid in NOS/VE COBOL. In a NOS/VE COBOL program, any sections numbered 50 or greater could function differently than with COBOL 5. The differences occur in altered GO TO statements (through the ALTER statement) and in PERFORM exits. If all PERFORM statements exit normally and the ALTER statement is not used, no conversion of the program is necessary; otherwise, manual conversion is necessary.

You can use the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph of the Environment Division to change the range designations, but the clause has no effect on the program.

## Termination Dump Facility

This facility does not exist in NOS/VE COBOL.

## Utility Routines

The names of utility routines provided for system interface have been either eliminated or changed. If only the name is changed, the old name is converted to the new name automatically during compilation.

Using BASE_LANGUAGE=COBOL5 in the COBOL command causes each utility name in a COBOL 5 program that exists to be converted to the new name and correct form for NOS/VE COBOL; however, this does not guarantee that the routine will execute properly.

The changes are as follows:

C.CMMMV Has been deleted. (CMM is not available.)

C.DMRST Has been deleted.

C.DTCMP CBP$DATE_COMPILED is the new name. The format is now installation dependent. Any processing that relied on the old format must be changed.

C.DSPDN CBP$DISPLAY_HEX_DATA is the new name. The contents of items are displayed in hexadecimal as well as in 7-bit ASCII format. Only one parameter is allowed. Because this routine is normally used only for debugging, few programs should be affected.

C.FILE Has been deleted. You can substitute with a call to CLP$SCAN_COMMAND_LINE specifying a NOS/VE command for setting file attributes.

C.GETEP CBP$GET_EXECUTION_PARAMETERS is the new name. The parameters are different in NOS/VE COBOL. To convert, change the call and the parameters.

C.IOENA CBP$IO_ERROR_NO_ABORT is the new name. The new routine functions the same as the C.IOENA routine.

C.IOST CBP$IO_STATUS is the new name. The status codes for BAM and AAM on NOS and NOS/BE differ for file interface routines on NOS/VE; therefore, each use of codes returned through this routine requires manual conversion.

C.LOK Has been deleted.

C.SEEK Has been deleted. No similar function exists.

C.SORTP Has been deleted. The sort under NOS/VE now provides the function.

C.UNLOK Has been deleted.

## Other Differences

The last group of differences between NOS/VE COBOL and COBOL 5.3 are discussed as follows:

Alphabet-name

Labels

Print records

Program-name

Reference modification

Switch-7 and switch-8

### Alphabet-Name

ASCII-64 and CDC-64 are alphabet names that have been deleted from NOS/VE COBOL. They are automatically converted to 7-bit ASCII and CDC, respectively. The use of different alphabet-names in collating sequences and code-sets has changed. See the discussions on these two areas for specific differences.

### Labels

Every file has a system label (where the attributes are stored), but no user label. The compiler allows label declarations in File Description entries, but labels are not processed at execution time.

The lack of labels should not cause any problems because almost no processing is performed in COBOL 5. Labels are currently ignored on mass storage.

### Print Records

Print files (such as PRINTF=YES, LINAGE, and so forth) no longer have the first character of the record area as a carriage control character. This should not impact any programs since this character could only be accessed by using SAME RECORD AREA and using a nonprint file record.

### Program-Name

The program-name in the PROGRAM-ID paragraph or the CALL statement can be a maximum of 30 characters in length. In NOS/VE COBOL, all characters up to the maximum of 30 are used; in COBOL 5, only the first seven characters are used.

### Reference Modification

The phrase END for unspecified length has been removed to conform to the proposed ANSI standard. This format omits the length altogether. To convert programs, remove the word END. END is still accepted by the compiler, however.

### SWITCH-7 and SWITCH-8

In COBOL 5, SWITCH-7 and SWITCH-8 are internal switches; in NOS/VE COBOL, these switches are external switches. To convert the program, change the switch numbers to any numbers in the range 9 through 136.

- Using BASE_LANGUAGE=COBOL5 in the compiler call causes SWITCH-7 and SWITCH-8 to be processed as SWITCH-135 and SWITCH-136, respectively. Since these switches are unavailable in COBOL 5, no conflict arises unless the switches are defined elsewhere in the NOS/VE COBOL program. The difference is in the execution speed and in the initial setting.

# Migrating APL Workspaces 16

# Migrating APL Workspaces 16

To migrate an APL2 workspace from NOS to NOS/VE, you must first convert the APL2 workspace into a form that can be used with APL for NOS/VE.

You must then use APL for NOS/VE to modify your APL2 functions if your functions use features that differ between APL2 and APL for NOS/VE. The following topics discuss the feature differences between the two implementations of APL:

| | |
|---|---|
| File-Related Differences | Special Functions |
| Workspace Constraints | New Features |
| Discontinued Features | Other Changes |

## Converting APL2 Workspaces and Files

The APL for NOS/VE product includes a supplied workspace that simplifies the conversion of APL2 workspaces and APL2 structured files from APL2 to APL for NOS/VE format. The name of this supplied workspace is:

```
:$SYSTEM.APL.CONVERT_APL2
```

This workspace converts the APL2 workspace or structured file to a form needed by APL for NOS/VE. Note that any feature differences between the two APL implementations are not handled by this workspace; these feature differences require changes by you.

The workspace or structured file to be converted can reside either on NOS or on NOS/VE.

You can transfer an APL2 workspace or structured file from NOS to NOS/VE using the GET_FILE command described elsewhere in this manual.

Before you transfer a structured file, you must remove all end-of-record marks from the file. You do this using the AFIFIX utility on NOS. The following illustrates how to transfer a structured file. While on NOS, enter:

```
GET,AFIFIX/UN=APL0.  <---------------------- Gets the AFIFIX utility on NOS.
AFIFIX,NOEOR,FILEA,FILEB.  <----------------- Removes end-of-record marks from structured
                                               files FILEA and FILEB; notice that the NOEOR
                                               parameter must be specified.
```

Now log on to NOS/VE and enter:

```
/get_file to=filea data_conversion=b60  <----- These commands transfer FILEA and FILEB from
/get_file to=fileb data_conversion=b60          NOS to NOS/VE; the DATA_CONVERSION=B60
                                               parameter must be specified to ensure proper
                                               conversion.
```

The following illustrates how to transfer a workspace. Logon to NOS/VE and enter:

```
/get_file to=ws1 data_conversion=b60  <------- Transfers workspace WS1 from NOS to NOS/VE.
                                               The DATA_CONVERSION=B60 parameter must be
                                               specified to ensure proper conversion.
```

To use :$SYSTEM.APL.CONVERT_APL2 to convert an APL2 workspace or structured file, first log in to NOS/VE and invoke the APL for NOS/VE system. Then load supplied workspace :$SYSTEM.APL.CONVERT_APL2; for example:

```
)LOAD :$SYSTEM.APL.CONVERT_APL2
```

This workspace is a self-starting workspace; therefore, when the workspace is loaded, it begins execution immediately. This is a dialogue-driven workspace that prompts you for each item of information needed to convert an APL2 workspace or structured file. Online help information is also available after you load this workspace.

## File-Related Differences

The major difference between APL2 and APL for NOS/VE is that APL for NOS/VE uses the more powerful NOS/VE file system. This means that any file references in your APL functions must be changed to conform to the NOS/VE syntax. Specifically:

- File names can be up to 31 characters long.
- Multiple level catalogs, which require a file path, are allowed.
- Indirect files do not exist.
- The concept of public, private, and semiprivate are replaced by an expanded file permit capability (SCL CREATE_FILE_PERMIT command).
- The concept of file cycles exists under NOS/VE.
- A NOS/VE file must be opened explicitly.
- Additional access and sharing modes for files are present under NOS/VE.

## Workspace Constraints

APL for NOS/VE removes some of the constraints on the size of various constructs that were present in APL2. This is possible because of the virtual memory capability of NOS/VE. The following is a summary:

- A saved workspace can be up to 2000 million bytes (250 million words); an active workspace can be larger than a saved workspace (up to 4000 million bytes in some cases). These constraints might differ at your installation.
- Workspaces can be shared by several users at one time. Any variables or functions that are changed by a particular user are placed in an individual segment for that user; each user does NOT have an individual copy of any unchanged variables and functions. The individual segment containing changed variables and functions can grow to 2000 million bytes; the unchanged part can be up to 2000 million bytes, too.
- Arguments to the execute function can be character vectors of up to 2000 million elements.
- Each line in a user-defined function, including the function header, can be character vectors of up to 2000 million characters.
- The APL system itself is shared among all APL users and does not require a portion of a user's memory allocation.

## Discontinued Features

The system functions $QDFI and $QDTM are no longer supported. Using these functions causes a message to be sent to your job log. The message provides you with the workspace name and function name where each unique reference occurs to help you locate all uses of these functions.

The FERASE function, which was related to direct vs. indirect access files on NOS, is no longer needed or supported.

Quad-PL can no longer be used to control paging for a CRT terminal. Paging is now performed by NOS/VE network commands.

The TERMINAL_TYPE parameter of the APL command has changed; see the TERMINAL_TYPE discussion in chapter 7 for more information.

The power function (LN*RN) no longer uses rational approximation when the left argument is negative and the right argument is not integral. A DOMAIN ERROR results.

The facilities provided by AFIFIX, AWSFIX, and AOVFIX are no longer needed or supported.

## Special Functions

APL for NOS/VE includes a number of functions called special functions. Special functions provide facilities that, in other APL systems, were provided by numerous system functions and the I function.

Like user-defined functions, special functions have names and can be copied and erased. Unlike user-defined functions, special functions are part of the APL system and are written in the implementation language for the APL system. The special functions are distributed in the supplied workspaces distributed with the APL for NOS/VE product. You cannot write your own special functions. If a change is made to a special function, that change is automatically propagated without your intervention to every workspace where it is used.

Special functions are provided primarily to give you access to files, the operating system, and so forth, and to provide you with faster execution. One special function that is particularly useful is the EXECUTE_SCL function, which enables you to execute SCL commands from within the APL system.

See the APL Language Definition Usage Manual for more information about special functions.

## New Features

The following is a summary of the new features provided in APL for NOS/VE:

- Shared variables are provided.
- An empty numeric vector constant can be written as $ZL.
- The last line entered in immediate execution mode can be modified and re-entered.
- When a user-controlled system parameter is localized, the parameter's initial value in the function is the same as its value before the function was entered. An IMPLICIT ERROR can no longer occur. Erasing a user-controlled system parameter causes the value of the parameter in the clear workspace (:$SYSTEM.APL.CLEARWS) to be used.
- Function definition mode and system commands are enhanced.
- $QDTRAP now allows you to specify which events are to be trapped and allows you to disable user interrupts.
- The printing width for $QP output is no longer controlled by $QDPW; the NOS/VE network checks the page width.
- Input received by $QP is always of rank 1. (In APL2, single characters were returned as scalars.)
- $QDPW sets the page width for your terminal when you enter APL. The page width is not changed when you load another workspace.

- APL workspaces and structured files now have file attributes that can be displayed and recognized outside of the APL system using NOS/VE utilities. These attributes are not lost when the file is copied.

- Error messages beginning with 20: have been changed to provide you with all information available from NOS/VE.

- Two new system commands are added. They are )QUIT, which is the same as )SYSTEM, and )RESET, which clears the state indicator list.

## Other Changes

Other APL changes are:

- Many nontrivial math functions return slightly different results because of slight differences in the CYBER 180 floating-point hardware.

- The output of monadic format differs slightly. Differences are due in part to the larger magnitude of numbers allowed by the CYBER 180. Also, the rules for when numbers are displayed in exponential form have been simplified.

- In dyadic format using exponential forms where APL decides the column width, more columns are reserved for the exponent.

- The definition of equality between two numeric values is now symmetrical in all cases. (In APL2, A=B could return 1 while B=A returned 0.) This change also affects less than, greater than, and so forth.

- The first element of the Account Information vector is always 0; the account name can be obtained using special function ACCOUNT_NAME, which is in workspace :$SYSTEM.APL.WSFNS.

- The )COPY system command requires a slash to separate the names to be copied from the other parameters.

- Special function FSTATUS returns 10 elements per file instead of 9.

- Errors 11:, 21:, 22:, and 24: are now returned as error 20:.

- CFREAD and CFWRITE use the full 256 character set.

- A new error message has been added. 25: AXIS ERROR indicates that the value given as a function index is not correct.

- The FNAMES function no longer returns a user number when a file from another catalog is accessed; FNAMES returns only the local file name.

- The FPACK function no longer accesses the file itself. The file to be packed must first be obtained using FTIE, FCREATE, and so forth.

- Catalogs APL1 and APL0 are combined into :$SYSTEM.APL.

- The $QDLIB function result differs from the result returned in APL2.

# Migrating Pascal Programs 17

# Migrating Pascal Programs 17

NOS/VE Pascal is similar to NOS Pascal Version 1. A NOS Pascal program that conforms to Standard Pascal should compile and execute correctly under NOS/VE Pascal.

However, certain extensions to Standard Pascal that are supported by NOS Pascal are not supported by NOS/VE Pascal. The following paragraphs discuss these extensions and other areas of difference between NOS Pascal and NOS/VE Pascal. For more information about the differences between NOS/VE Pascal and NOS Pascal Version 1, see the Pascal Usage manual.

## Predefined Routines

The following predefined routines are provided by NOS Pascal but not by NOS/VE Pascal:

```
EXPO(a)

UNDEFINED(a)
```

In addition, the following form of the TRUNC routine is not provided by NOS/VE Pascal:

```
TRUNC(a,n)
```

The form TRUNC(a) is provided by NOS/VE Pascal.

## Segmented File Operations

The following predefined routines and functions for operating on segmented files are supported by NOS Pascal but not by NOS/VE Pascal:

```
EOS(f)

GETSET(f,n)

PUTSEG(f,n)
```

In addition, the following form of the REWRITE routine is provided by NOS Pascal but not by NOS/VE Pascal:

```
REWRITE(f,n)
```

The form REWRITE(f) is supported by NOS/VE Pascal.

## Type ALFA

Type ALFA is a machine-dependent type that allows a string type that is one CYBER 170 word in length. This type is not meaningful on a CYBER 180, and is therefore not supported by NOS/VE Pascal.

## EXTERNAL Directive

In NOS/VE Pascal, the FORTRAN directive is not supported, and the EXTERN directive is replaced by the EXTERNAL directive.

## Compiler Directives

NOS/VE Pascal does not support compiler directives.

## Value Initialization

NOS and NOS/VE do value initialization differently. NOS supports a replication factor to facilitate initialization of structured variables. NOS/VE provides the same capability by using structured constants, which have a different syntax for replication. See the Pascal Usage manual for more information.

## Strings

NOS/VE Pascal provides more powerful string processing capabilities than NOS Pascal. NOS/VE Pascal allows both fixed-string and variable-string types. Either type can be up to 65535 characters long. The length of a variable-string type is determined dynamically. The following operations can be performed on strings:

**Assignment**

**Relational**

**Concatenation**

NOS/VE Pascal provides the following predefined functions for operating on strings:

| | |
|---|---|
| INDEX | Returns the starting position of a substring within a string. |
| LENGTH | Returns the length of a specified string. |
| MAXLENGTH | Returns the maximum length of a variable-string. |
| SUBSTR | Returns a substring of a specified string. |

In NOS/VE Pascal, strings can be passed as actual arguments, and they can be read from or written to a text file.

## Collating Sequence

To do comparisons, Pascal on NOS/VE uses a collating sequence different from that used by Pascal on NOS. The NOS/VE Pascal uses the ASCII collating sequence (listed in appendix C of the Pascal Usage manual). This means that the results of comparisons can differ between NOS and NOS/VE.

Comparisons that yield the same results in NOS and NOS/VE include:

Alphabetic character data compared to alphabetic character data

Comparisons of numeric strings

Tests of equality

Tests of inequality (<>)

Comparisons that yield different results in NOS and NOS/VE include:

Comparisons of alphanumeric data

Special character data compared to special character data

Comparisons using ordinal type operands (because ORD(character_expression) provides a different value in NOS/VE than in NOS)

## PASCAL Command

The NOS/VE PASCAL command has the following format:

```
PASCAL
    INPUT=file
    BINARY=file
    LIST=file
    DEBUG_AIDS=list of keyword
    ERROR=file
    ERROR_LEVEL=keyword
    LIST_OPTIONS=list of keyword
    OPTIMIZATION_LEVEL=list of keyword
    RUNTIME_CHECKS=list of keyword
    STANDARDS_DIAGNOSTICS=list of keyword
    TERMINATION_ERROR_LEVEL=keyword
    STATUS=status variable
```

### INPUT or I

This parameter specifies the file containing the source code to be compiled.

| | |
|---|---|
| Omitted | Same as I=$INPUT. |
| I=file | Source code is read from the specified file. |

### BINARY or B

This parameter specifies the file to receive the compiled object code.

| | |
|---|---|
| Omitted | Same as B=$LOCAL.LGO. |
| B=file | Binary object code is written to the specified file. |

### LIST or L

This parameter specifies the file to receive the compiler output listing. Output listing options are selected by the LIST_OPTIONS parameter.

| | |
|---|---|
| Omitted | Same as LIST=$LIST (no listing is produced). |
| LIST=file | Output listing is written to the specified file. |

## DEBUG_AIDS or DA

This parameter selects debugging options.

| | |
|---|---|
| Omitted | Same as DA=NONE. |
| DA=DT | Generates line number, symbol tables, and source map loader tables for use by the Debug facility. |
| DA=NONE | Inhibits generation of line number and symbol tables. |
| DA=ALL | Same as DA=DT. |

## ERROR or E

This parameter specifies a file to receive compile-time error messages.

| | |
|---|---|
| Omitted | Same as ERROR=$ERRORS. |
| ERROR=file | Error messages are written to the specified file. |

## ERROR_LEVEL or EL

This parameter specifies the severity level of errors to be listed.

| | |
|---|---|
| Omitted | Same as ERROR_LEVEL=W. |
| ERROR_LEVEL=W | Warning (informative) errors and errors of levels F and C are listed. |
| ERROR_LEVEL=F | Fatal errors and errors of level C are listed. |
| ERROR_LEVEL=C | Only catastrophic errors are listed. |

## LIST_OPTIONS or LO

This parameter selects compiler output listing options.

| | |
|---|---|
| Omitted | Same as LIST_OPTIONS=S. |
| LIST_OPTIONS=A | Lists attributes of all entities defined in the source program. |
| LIST_OPTIONS=O | Lists object code. |
| LIST_OPTIONS=R | Lists cross-reference map of symbolic names used in the source program. |
| LIST_OPTIONS=S | Lists source code. |
| LIST_OPTIONS=NONE | No listing options are selected. |

Multiple options can be selected by separating the options by a space or comma and enclosing them in parentheses. For example:

```
LIST_OPTIONS=(O,R,S,A)
```

## OPTIMIZATION_LEVEL or OL

This parameter selects the level of optimization performed by the compiler.

| | |
|---|---|
| Omitted | Same as OL=LOW. |
| OL=DEBUG | Object code is formatted for use by the Debug facility. |
| OL=LOW | Minimum optimization is performed. |

## RUNTIME_CHECKS or RC

This parameter selects runtime error checking options.

| | |
|---|---|
| Omitted | Same as RUNTIME_CHECKS=NONE. |
| RUNTIME_CHECKS=F | Selects checking of errors involving file variables and buffer variables. |
| RUNTIME_CHECKS=N | Selects checking for invalid use of pointer variables and NEW and DISPOSE procedures. |
| RUNTIME_CHECKS=R | Selects range checking for subranges, set assignments, and case variables. |
| RUNTIME_CHECKS=S | Selects array subscript bound checking. |
| RUNTIME_CHECKS=ALL | Selects all of the above runtime checks. |
| RUNTIME_CHECKS=NONE | Selects no runtime checks. |

Multiple options can be specified by separating the options by a space or comma and enclosing them in parentheses. For example:

```
RUNTIME_CHECKS=(R,S)
```

## STANDARDS_DIAGNOSTICS or SD

This parameter selects checking for use of nonstandard extensions in the source program.

| | |
|---|---|
| Omitted | Same as SD=NONE. |
| SD=W | Same as SD=(W,ISO). |
| SD=F | Same as SD=(F,ISO). |
| SD=(W,ISO) | Non-ISO usages are diagnosed as warning errors. |
| SD=(F,ISO) | Non-ISO usages are diagnosed as fatal errors. |
| SD=(W,ANSI) | Non-ANSI usages are diagnosed as warning errors. |
| SD=(F,ANSI) | Non-ANSI usages are diagnosed as fatal errors. |
| SD=NONE | Nonstandard usages are not diagnosed. |

## TERMINATION_ERROR_LEVEL or TEL

This parameter specifies the diagnostic severity level for which the compiler returns abnormal status.

| | |
|---|---|
| Omitted | Same as TEL=F. |
| TEL=W | Abnormal status is returned for all warning and higher level errors. |
| TEL=F | Abnormal status is returned for all fatal and catastrophic errors. |
| TEL=C | Abnormal status is returned for catastrophic errors only. |

## STATUS

This parameter specifies the name of an SCL status variable to receive error status code returned by the compiler.

| | |
|---|---|
| Omitted | Status code is returned to the caller of the Pascal command. |
| STATUS=variable | Status code is returned in the specified SCL status variable. |

# Appendixes

# Glossary for NOS/VE Use A

The definitions in this glossary apply to NOS/VE only, unless the individual definition states otherwise.

## A

Abort

The immediate abnormal termination of command or a task.

Access Mode

The manner in which records can be inserted into or retrieved from a file. Access mode can be sequential, random, or dynamic, depending on the ACCESS MODE clause. Access mode, open mode, and file organization affect subsequent operations.

Alphabetic Character

One of the following letters:

A through Z

a through z

See also Character and Alphanumeric Character.

Alphanumeric Character

An alphabetic character or a digit. See also Character, Alphabetic Character, and Digit.

ANSI Standard Language

The language defined by the American National Standards Institute. For COBOL, it is defined in American National Standard X3.23-1974. For FORTRAN, it is defined in American National Standard X3.9-1978.

ASCII

American National Standard Code for Information Interchange. A 7-bit code representing a prescribed set of characters. The 7-bit ASCII code character is stored right-justified in an 8-bit byte.

Assignment Statement

In computer languages, a statement that assigns a value to a variable. In FMU, a statement that reads a value from an input record, converts the value, and copies the value to an output record.

Attach

The process of retrieving a permanent file for use by a job. The process involves specifying the proper file identification and, if necessary, password.

## B

Batch Mode

A mode of execution where a job is submitted and processed as a unit with no intervention from the user. Contrast with Interactive Mode.

Beginning-of-Information (BOI)

The file boundary that marks the beginning of the file.

Bit

A binary digit. A bit has the value 0 or 1. See also Byte.

BOI

See Beginning-of-Information.

Boolean

A kind of value that is evaluated as TRUE or FALSE.

Boolean String Constant

In NOS/VE FORTRAN, a constant that represents a string of one through eight characters. It is equivalent to a Hollerith constant in NOS FORTRAN 5.

Buffer Statement

One of the input/output statements BUFFER IN or BUFFER OUT.

Byte

A group of contiguous bits. For NOS/VE, one byte is equal to 8 bits. An ASCII character code uses the right-most 7 bits of one byte.

Byte Addressable File Organization

A file organization in which records are accessed by their byte address.

A file described by the ORGANIZATION IS BYTE-ADDRESS clause. A byte addressable file is characterized by records identified by a key that indicates the relative byte within a mass storage file at which the record begins.

Byte Offset

A number corresponding to the number of bytes beyond the beginning of a line, procedure, module, or section.

## C

Catalog

A directory of files and catalogs maintained by the system for a user. The catalog $LOCAL contains only file entries.

The part of a path that identifies a particular catalog in a catalog hierarchy. The format is as follows:

```
name.name...name
```

where each name is a catalog. See also Catalog Name and Path.

Catalog Name

The name of a catalog in a catalog hierarchy (path). By convention, the name of the user's master catalog is the same as the user's user name.

Character

A letter, digit, space, or symbol that is represented by a code in one or more of the standard character sets.

It is also referred to as a byte when used as a unit of measure to specify block length, record length, and so forth.

A character can be a graphic character or a control character. A graphic character is printable; a control character is nonprintable and is used to control an input or output operation.

Collated Key

A key consisting of 1 through 255 8-bit characters. These keys are sorted according to the sequence indicated by the user-specified collation table in effect. Contrast with Uncollated Key.

Collating Sequence

The sequence in which characters are ordered for purposes of sorting, merging, and comparing.

Collation Table

A data structure that orders a set of characters. The character order is used when sorting keys in an indexed sequential file.

Command

A statement that initiates a specific operation on NOS/VE. A command name is recognized by the SCL interpreter if it appears as an entry in the command list.

Command List

One or more entries that define the commands that are currently available. A command list is an object library, catalog, or special entry $SYSTEM.

Comment

A line or sequence of characters that is not interpreted or compiled and is for documentary purposes only.

Compilation Time

The time at which a source program is translated by the compiler to an object program that can be loaded and executed. Contrast with Execution Time.

Compiler

A processor that accepts source code as input and generates object code as output.

Condition Code

Alphanumeric characters that uniquely identify a NOS/VE diagnostic. The condition code is returned as part of the status record when an abnormal status occurs.

Control Statement

A statement used to structure and control the flow of a job.

Current Position

The line in the current deck from which the editor determines the location for an operation. The current position line can be referenced with the keyword CURRENT.

Cycle

A numbered version of a file that can be registered with the same file name and in the same catalog as the other versions of the file. Each permanent file can have from 1 through 999 cycles.

See also Cycle Number and Cycle Reference.

Cycle Number

An unsigned integer from 1 through 999 that identifies a specific version of a permanent file.

Cycle Reference

The cycle of a permanent file to be accessed. A cycle reference can be either an unsigned integer or one of the following designators:

```
$HIGH
$LOW
$NEXT
```

## D

Data Block

A logical or physical grouping of records in which user records are stored in an indexed sequential file.

Deck

A sequence of lines in a source library that can be manipulated as a unit by the Source Code Utility (SCU).

Default

The system-defined value assumed in the absence of a user-specified value.

Delimiter

An indicator that separates and organizes data.

Digit

One of the following characters:

0 1 2 3 4 5 6 7 8 9

Direct-Access File Organization

A keyed-file organization in which each record is accessed directly through its hashed primary-key value. Records can be accessed sequentially, but the records are not returned in sorted order. Contrast with Indexed-Sequential File Organization.

Direct Access Input/Output

A method of input/output in which records can be read or written in any order.

Directive

A directive consists of a directive name followed by a parameter list and is specified in a directive file.

Directive File

A file that contains only directives.

Display Code

A 64-character subset of the ASCII code, which consists of alphabetic letters, symbols, and numerals. This character set is not used by NOS/VE.

Dual State

State in which two operating systems execute simultaneously on the same mainframe. Currently, NOS/VE and either NOS version 2 or NOS/BE Version 1.5 are such partners.

Dynamic Access

An access mode that allows a nonsequential file on mass storage to be accessed randomly or sequentially, depending on the format of the access statement.

## E

EBCDIC

The abbreviation for extended binary-coded decimal interchange code, an 8-bit code representing a coded character set.

Ellipsis

1. Two or more consecutive periods at the end of a physical line to indicate command line continuation. The ellipsis can be optionally preceded and/or followed by a space.

2. Two consecutive periods separating two values to indicate a range of values in a parameter list. See also Value Element.

Embedded Key

A key that is contained within a record, as opposed to a key that is defined outside of a record (such as in the Working-Storage or Common-Storage sections of a COBOL program).

End-of-Information (EOI)

The point at which data in the file ends.

End-of-Partition (EOP)

A special delimiter in a file with variable record type.

EOI

See End-of-Information.

Epilog

The SCL statement list that is executed at the end of a job.

Execution Time

The time at which a compiled source program is executed. Also known as Run Time.

# F

F-Type Record

Fixed-length records, as defined by the ANSI Standard.

Family

A logical grouping of NOS/VE users that determines the location of their permanent files. A family can be subdivided into accounts and projects.

Family Name

A name that identifies a NOS/VE family. See Family.

Field Descriptor

An FMU element that describes a data field in an input or output record in terms of data type, starting position, and field length.

File

1. A collection of information referenced by a name.
2. An SCL element specifying a temporary or permanent file, including its path and, optionally, a cycle reference (for permanent files). A file is identified by specifying a path and, optionally, a cycle reference (for permanent files) as follows:

   path.cycle reference

File Attribute

A characteristic of a file. Each file has a set of attributes that completely defines the file structure and processing limitations.

File Information Table

The internal table through which a FORTRAN program communicates with the FORTRAN file interface routines.

File Management Utility (FMU)

A file management utility that can manipulate data from an input file and create output files formatted according to specifications.

File Name

The name of a NOS/VE file. It is used in a file reference to identify the file. See also Name.

File Organization

Defines the way records are stored in a file. The available file organizations are sequential, byte addressable, direct access, and indexed sequential.

File Position

The location in the file at which the next read or write operation will begin. The file position designators are:

| | |
|---|---|
| $ASIS | Leave the file in its current position. |
| $BOI | Position the file at the beginning-of-information. |
| $EOI | Position the file at the end-of-information. |

File Reference

An SCL element that identifies a file and, optionally, the file position to be established prior to use. The format of a file reference is as follows:

file.file position

where the file positions are:

$BOI

$ASIS

$EOI

See also File and File Position.

Floating-Point Number

A method of internal binary representation for numbers written with a decimal point; corresponds to a FORTRAN REAL or COBOL COMPUTATIONAL-1 number. Also can be stored as double precision, corresponding to FORTRAN DOUBLE PRECISION or COBOL COMPUTATIONAL-2. Also can be stored as complex, corresponding to FORTRAN COMPLEX.

FMU

See File Management Utility.

FSE

See Full Screen Editor.

Full Screen Editor

A NOS/VE editor that enables you to edit files screen-by-screen.

## H

Hollerith Constant

In FORTRAN 5, a constant that represents a string of one through eight characters. Hollerith constants are equivalent to boolean string constants in NOS/VE FORTRAN.

## I

Index Block

A block in an indexed sequential file that contains ordered keys and pointers to index blocks or other data blocks. Contrast with Data Block.

Index Record

An internal record in an index block that guides the system in locating data records by primary key value. An index record consists of a primary key value and a pointer to a block. The primary key value in the index record matches the primary key value of the first record in either a lower-level index block or a data block.

Indexed-Sequential File Organization

A keyed-file organization in which each record is accessed by finding its primary-key value in the file index. When records are read sequentially from an indexed sequential file, they are returned sorted by primary-key value. Contrast with Direct-Access File Organization.

Integer

Numeric data (positive or negative) that does not have any digits to the right of the assumed decimal point. An integer is stored internally as a binary value rather than a character value.

Integer Constant

One or more digits and, for hexadecimal integer constants, the following characters

A B C D E F a b c d e f

A hexadecimal integer constant must begin with a digit. A preceding sign and subsequent radix are optional.

Integer Key

A signed binary key used with an indexed sequential file. Integer keys are sorted by arithmetic value.

Interactive Mode

A mode of execution where a user enters commands at a terminal and each command elicits a response from the computer. Contrast with Batch Mode.

## J

Job

A set of tasks executed for a user name. NOS/VE accepts interactive and batch jobs.

Job Log

A chronological listing of all operations associated with a terminal session or batch job from login to logout.

## K

Key

A string of characters or a number that the user defines to uniquely identify a record.

Keyword

A parameter value that has special meaning in the context of a particular parameter. For example, a parameter called COUNT might normally expect an integer but could be given the keyword value ALL.

## L

Line

A sequence of characters SCU recognizes as a record of text.

Load Time

The time at which an object program is loaded into memory and prepared for execution.

Loader

The system software that loads a compiled object program into memory and prepares it for execution.

Local File

A file that is accessed via the local catalog named $LOCAL. See also File, Path, and Local Path.

Local File Name

The name used by an executing job to reference a file while the file is assigned to the job's $LOCAL catalog. Only one file can be associated with a given name in one job; however, in one job a file can have more than one instance of open by that name.

Local Path

Identifies a local file as follows:

```
$LOCAL.file name
```

Login

The process used to gain access to the system.

Logout

The process used to end a terminal session.

## M

Mass Storage

Disk storage that allows random file access and permanent file storage.

Mass Storage Input/Output

The type of input/output used for random access to files; it involves the subroutines OPENMS, READMS, WRITMS, CLOSMS, and STINDX.

Master Catalog

The catalog the system creates for each new user name. The master catalog contains entries for all permanent files and catalogs a user creates. By convention, the name of the master catalog is the same as the user name.

Megabyte (MB)

A group of bytes. One megabyte represents 1,048,576 bytes.

Module

A unit of text accepted as input by the loader or object library generator.

## N

Name

In SCL, a combination of from 1 through 31 characters chosen from the following set:

Alphabetic characters (A through Z and a through z)

Digits (0 through 9)

Special characters (#, @, $, [, ], \, ^, `, {, |, }, ~, or _)

The first character of a name cannot be a digit.

Name Call

A method of obtaining and executing a program in which you enter the name of the file containing the object program.

Native Character Set

The character set associated with the operating system, which is the ASCII character set as defined by the standard ANSI X3.4-1977.

Nonembedded Key

A primary key that is not physically contained in the record. Internally, a nonembedded key is stored before its record in a data block.

NOS/VE

Acronym for Network Operating System/Virtual Environment, an operating system for the host computer. NOS/VE has all of the capabilities of NOS and NOS/BE. In addition, NOS/VE uses virtual memory.

## O

Object Code

Executable code produced by a compiler.

Object File

A file containing one or more object modules.

Open

A set of preparatory operations performed on a file before input and output can occur.

Optimization

The manipulation of object code to reduce execution time. The level of optimization performed by the compiler can be selected through the OPTIMIZATION_LEVEL parameter on the FORTRAN command.

## P

Packed Decimal

A numeric data format where each digit is represented by four bits, with two digits per standard 8-bit byte. In COBOL, this data is described as PACKED DECIMAL or COMPUTATIONAL-3.

Padding

Space deliberately left unused. Fixed length records are padded if the data provided for the record is shorter than the record length. Within keyed files, blocks are padded during file creation to allow easy addition or expansion during later file updates.

Page

An allocatable unit of real memory.

Parameter

A value list optionally preceded by and equivalenced to a parameter name. For example:

```
parameter name = value list  or  value list
```

Parameter List

A series of parameters separated by spaces or commas.

Parameter Name

A name that uniquely identifies a parameter.

Parameter Value

See Value.

Partition

A unit of data on a sequential or byte addressable file delimited by end-of-partition separators or the beginning- or end-of-information.

Path

Identifies a file. It may include the family name, user name, subcatalog name or names, and file name.

Permanent Catalog

A catalog of permanent files.

Permanent File

A mass storage file preserved by NOS/VE across job executions and system deadstarts. A permanent file has an entry in a permanent catalog. See also File.

Position-Dependent Parameter

A parameter that must appear in a specified location, relative to other parameters. Contrast with Position-Independent Parameter.

Position-Independent Parameter

Parameter that consists of a parameter name followed by a value list. Contrast with Position-Dependent Parameter.

Primary Key

The required key in a keyed file. The primary-key value must be defined for a file when the file is first created, and each record in the file must have a unique value for the key.

Procedure

A sequence of SCL commands executed when the procedure name is entered. It can be stored as a module on an object library.

Program Attribute

A characteristic of a program as defined in the program description or by a job default value.

Prolog

The SCL statement list that is executed at the beginning of each job.

## R

Radix

Specifies the base of a number. NOS/VE recognizes number bases from base 2 through base 16. A radix enclosed in parentheses must follow a nondecimal number.

See Integer Constant.

Random Access

The process of reading or writing a record in a file without having to read or write the preceding records; applies only to mass storage files. Contrast with Sequential Access.

Random File Organization

A file with byte addressable, direct access, indexed sequential, or relative (COBOL) organization in which individual records can be accessed by the value of their keys. In FORTRAN, a random file has byte addressable organization and is processed by mass storage subroutines. Contrast with Sequential File Organization.

Range

A value represented as two values separated by an ellipsis. The element is associated with the values from the first value through the second value. The first value must be less than or equal to the second value. For example:

```
value..value
```

Real State

The NOS or NOS/BE operating system in a dual state environment. Contrast with Virtual State.

Record

A unit of data that can be read or written by a single I/O request. Also a set of related data processed as a unit when reading or writing a file.

Record Length

The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Relative File

A file described by the ORGANIZATION IS RELATIVE clause. A relative file is characterized by fixed-length records with key values equivalent to the ordinal positions of records in the file.

Rewind

An operation that positions a file at the beginning-of-information.

Run Time

See Execution Time.

## S

SCL

See System Command Language.

SCL Procedure

A sequence of SCL commands executed when the procedure name is entered. It can be stored as a file or as a module on an object library.

SCL Statement

The basic unit of input that is interpreted by SCL. The following are the SCL statement types:

Command
Assignment statement
Control statement

SCL Variable

The means of storing a value to be tested or displayed by an SCL statement.

Sequential Access

The processing of records in order (physical or logical). Contrast with Random Access.

Sequential File Organization

A file with records stored and retrieved in the order in which they were written. No logical order exists other than the relative physical record position.

Source Code

Statements written for input to a compiler.

Source Listing

A compiler-produced listing of the user's original source program.

Source Program

A set of statements written in a programming language such as FORTRAN and COBOL.

Standard File

A file that provides a default file for use by job files and other files. The standard files are identified by the following names:

| | |
|---|---|
| $COMMAND | $COMMAND_OF_CALLER |
| $ECHO | $ERRORS |
| $INPUT | $LIST |
| $OUTPUT | $RESPONSE |

Statement

See SCL Statement.

Status Variable

A variable record of kind status that holds the completion status of a command.

String

A value that represents a sequence of characters.

String Constant

A sequence of characters delimited by apostrophes (´). An apostrophe can be included in the string by specifying two consecutive apostrophes.

Subcatalog

A catalog registered in the master catalog or another subcatalog. See also Catalog.

System Command Language (SCL)

The block-structured interpretive language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

## T

Task

The instance of execution of a program.

## U

U-Type Record

A record for which the record structure is undefined.

Uncollated Key

A key consisting of from 1 through 255 8-bit characters. These keys are sorted by the magnitude of their binary ASCII code values. Contrast with Collated Key.

User Name

A name that identifies a NOS/VE user and the location of a user´s permanent files in the user´s family.

User Path

Identifies a file or catalog via a user name and optionally a relative path as follows:

.user name.relative path or $USER.relative path

## V

V-Type Record

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

Value

An expression or application value specified in a parameter list. Each value must match the defined kind of value for the parameter. Keywords, constants, and variable references are all values.

Value Element

A single value or a range of values represented by two values separated by an ellipsis. For example:

value or value..value

See also Value, Value List, and Value Set.

Value List

A series of value sets separated by spaces or commas and enclosed in parentheses. If only one value set is given in the list, the parentheses can be omitted. For example:

(value set,value set,value set) or value set

See also Value, Value Element, and Value Set.

Value Set

A series of value elements separated by spaces or commas and enclosed in parentheses. If only one value element is given in the set, the parentheses can be omitted. For example:

(value element,value element,value element) or value element

See also Value, Value Element, and Value List.

Variable

A language element that represents a data value that can be changed during execution.

Variable Name

A name that identifies a variable.

Virtual Memory

The memory access mode using virtual addresses. Each task uses the same virtual address range; the system associates the virtual address range referenced by a task to the physical memory assigned to the task.

Virtual State

The NOS/VE operating system in a dual state environment. Contrast with Real State.

## W

Word

A unit of measure for real memory. One word is 8 bytes or 64 bits of memory.

Working Catalog

The catalog used if no other catalog is specified on a file reference.

# RELATED MANUALS B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the SCL Language Definition Usage manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/explain
```

## Ordering Printed Manuals

Control Data manuals are available from your local Control Data office. Sites within the United States can also order manuals directly from Literature and Distribution Services at the following address:

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

When ordering a manual, please specify the complete title, publication number, and revision level. Please see the Literature and Distribution Services (LDS) catalog for further information on ordering manuals. You can obtain the LDS catalog from LDS.

## Accessing Online Manuals

To access an online manual, log in to NOS/VE and specify the online manual title (listed in table B-1) on the EXPLAIN command. For example, to read the FORTRAN online manual, enter:

```
/explain  manual=fortran
```

Table B-1. Related Manuals

| Manual Title | Publication Number | Online Title |
|---|---|---|
| **This Manual:** | | |
| Migration From NOS to NOS/VE Tutorial/Usage | 60489503 | MIGRATE_NOS |
| **Other Migration Manuals:** | | |
| Migration From NOS to NOS/VE Standalone Tutorial/Usage | 60489504 | |
| Migration From NOS/BE to NOS/VE Tutorial/Usage | 60489505 | MIGRATE_NOSBE |
| Migration From NOS/BE to NOS/VE Standalone Tutorial/Usage | 60489506 | |
| Migration From IBM to NOS/VE Tutorial/Usage | 60489507 | MIGRATE_IBM |
| Migration From VAX/VMS to NOS/VE Tutorial/Usage | 60489508 | MIGRATE_VAX |
| **SCL Manuals:** | | |
| SCL for NOS/VE Language Definition Usage | 60464013 | |
| SCL for NOS/VE System Interface Usage | 60464014 | |
| SCL for NOS/VE Quick Reference | 60464018 | SCL |
| SCL for NOS/VE Source Code Management Usage | 60464313 | |
| SCL for NOS/VE Object Code Management Usage | 60464413 | |
| SCL for NOS/VE Advanced File Management Usage | 60486413 | AFM |

(Continued on Next Page)

Table B-1. Related Manuals

(Continued From Previous Page)

| Manual Title | Publication Number | Online Title |
|---|---|---|
| **NOS/VE Product Descriptions:** | | |
| CYBIL for NOS/VE Language Definition Usage | 60464113 | |
| Pascal for NOS/VE Usage | 60485613 | PASCAL |
| APL for NOS/VE Language Definition Usage | 60485813 | |
| FORTRAN for NOS/VE Language Definition Usage | 60485913 | |
| FORTRAN for NOS/VE Quick Reference | L60485918 | FORTRAN |
| COBOL for NOS/VE Usage | 60486013 | COBOL |
| **Additional NOS/VE Manuals:** | | |
| Remote Host Facility Usage | 60460620 | |
| NOS/VE File Editor Tutorial/Usage | 60464015 | |
| CYBIL for NOS/VE File Interface Usage | 60464114 | |
| CYBIL for NOS/VE Keyed-file and Sort/Merge Interfaces Usage | 60464117 | |
| Diagnostic Messages for NOS/VE Usage | 60464613 | MESSAGES |
| NOS/VE System Information | L60488103 | default |
| Debug for NOS/VE Usage | 60488213 | |
| Debug for NOS/VE Quick Reference | L60488218 | DEBUG |
| CONTEXT for NOS/VE Usage | 60488403 | CONTEXT |

(Continued on Next Page)

Table B-1. Related Manuals

(Continued From Previous Page)

| Manual Title | Publication Number | Online Title |
|---|---|---|
| Additional Related Manuals: | | |
| NOS Version 2 Reference Set Volume 3 | 60459680 | |
| CYBER Interactive Debug Reference Manual | 60481400 | |
| CYBER Record Manager Basic Access Methods Reference Manual | 60495700 | |
| FORM Version 1 Reference Manual | 60496200 | |
| COBOL Version 5 Reference Manual | 60497100 | |
| CYBER Record Manager Advanced Access Methods Reference Manual | 60499300 | |

# Character Sets and Collating Sequences C

Table C-1 lists the ASCII (7-bit ASCII code) character set and collating sequence (the default sequence for NOS/VE).

The listings of predefined collation tables appear as follows:

| | |
|---|---|
| Table C-2 | OSV$ASCII6_FOLDED |
| Table C-3 | OSV$ASCII6_STRICT |
| Table C-4 | OSV$COBOL6_FOLDED |
| Table C-5 | OSV$COBOL6_STRICT |
| Table C-6 | OSV$DISPLAY63_FOLDED |
| Table C-7 | OSV$DISPLAY63_STRICT |
| Table C-8 | OSV$DISPLAY64_FOLDED |
| Table C-9 | OSV$DISPLAY64_STRICT |
| Table C-10 | OSV$EBCDIC |
| Table C-11 | OSV$EBCDIC6_FOLDED |
| Table C-12 | OSV$EBCDIC6_STRICT |

Table C-1. Full ASCII Character Set and Collating Sequence

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 0 | 00 | NULL | Null |
| 1 | 01 | SOH | Start of heading |
| 2 | 02 | STX | Start of text |
| 3 | 03 | ETX | End of text |
| 4 | 04 | EOT | End of transmission |
| 5 | 05 | ENQ | Enquiry |
| 6 | 06 | ACK | Acknowledge |
| 7 | 07 | BEL | Bell |
| 8 | 08 | BS | Backspace |
| 9 | 09 | HT | Horizontal tabulation |
| 10 | 0A | LF | Line feed |
| 11 | 0B | VT | Vertical tabulation |
| 12 | 0C | FF | Form feed |
| 13 | 0D | CR | Carriage return |
| 14 | 0E | SO | Shift out |
| 15 | 0F | SI | Shift in |
| 16 | 10 | DLE | Data link escape |
| 17 | 11 | DC1 | Device control 1 |
| 18 | 12 | DC2 | Device control 2 |
| 19 | 13 | DC3 | Device control 3 |
| 20 | 14 | DC4 | Device control 4 |
| 21 | 15 | NAK | Negative acknowledge |
| 22 | 16 | SYN | Synchronous idle |
| 23 | 17 | ETB | End of transmission block |
| 24 | 18 | CAN | Cancel |
| 25 | 19 | EM | End of medium |
| 26 | 1A | SUB | Substitute |
| 27 | 1B | ESC | Escape |
| 28 | 1C | FS | File separator |
| 29 | 1D | GS | Group separator |
| 30 | 1E | RS | Record separator |
| 31 | 1F | US | Unit separator |
| 32 | 20 | SP | Space |
| 33 | 21 | ! | Exclamation point |
| 34 | 22 | " | Quotation marks |
| 35 | 23 | # | Number sign |
| 36 | 24 | $ | Dollar sign |
| 37 | 25 | % | Percent sign |
| 38 | 26 | & | Ampersand |
| 39 | 27 | ' | Apostrophe |
| 40 | 28 | ( | Opening parenthesis |
| 41 | 29 | ) | Closing parenthesis |
| 42 | 2A | * | Asterisk |
| 43 | 2B | + | Plus |
| 44 | 2C | , | Comma |
| 45 | 2D | - | Hyphen |
| 46 | 2E | . | Period |
| 47 | 2F | / | Slant |
| 48 | 30 | 0 | Zero |
| 49 | 31 | 1 | One |
| 50 | 32 | 2 | Two |

(Continued on next page)

Table C-1. Full ASCII Character Set and Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 51 | 33 | 3 | Three |
| 52 | 34 | 4 | Four |
| 53 | 35 | 5 | Five |
| 54 | 36 | 6 | Six |
| 55 | 37 | 7 | Seven |
| 56 | 38 | 8 | Eight |
| 57 | 39 | 9 | Nine |
| 58 | 3A | : | Colon |
| 59 | 3B | ; | Semicolon |
| 60 | 3C | < | Less than |
| 61 | 3D | = | Equal to |
| 62 | 3E | > | Greater than |
| 63 | 3F | ? | Question mark |
| 64 | 40 | @ | Commercial at |
| 65 | 41 | A | Uppercase A |
| 66 | 42 | B | Uppercase B |
| 67 | 43 | C | Uppercase C |
| 68 | 44 | D | Uppercase D |
| 69 | 45 | E | Uppercase E |
| 70 | 46 | F | Uppercase F |
| 71 | 47 | G | Uppercase G |
| 72 | 48 | H | Uppercase H |
| 73 | 49 | I | Uppercase I |
| 74 | 4A | J | Uppercase J |
| 75 | 4B | K | Uppercase K |
| 76 | 4C | L | Uppercase L |
| 77 | 4D | M | Uppercase M |
| 78 | 4E | N | Uppercase N |
| 79 | 4F | O | Uppercase O |
| 80 | 50 | P | Uppercase P |
| 81 | 51 | Q | Uppercase Q |
| 82 | 52 | R | Uppercase R |
| 83 | 53 | S | Uppercase S |
| 84 | 54 | T | Uppercase T |
| 85 | 55 | U | Uppercase U |
| 86 | 56 | V | Uppercase V |
| 87 | 57 | W | Uppercase W |
| 88 | 58 | X | Uppercase X |
| 89 | 59 | Y | Uppercase Y |
| 90 | 5A | Z | Uppercase Z |
| 91 | 5B | [ | Opening bracket |
| 92 | 5C | \ | Reverse slant |
| 93 | 5D | ] | Closing bracket |
| 94 | 5E | ^ | Circumflex |
| 95 | 5F | _ | Underline |
| 96 | 60 | ` | Grave accent |
| 97 | 61 | a | Lowercase a |
| 98 | 62 | b | Lowercase b |
| 99 | 63 | c | Lowercase c |
| 100 | 64 | d | Lowercase d |

(Continued on next page)

Table C-1. Full ASCII Character Set and Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 101 | 65 | e | Lowercase e |
| 102 | 66 | f | Lowercase f |
| 103 | 67 | g | Lowercase g |
| 104 | 68 | h | Lowercase h |
| 105 | 69 | i | Lowercase i |
| 106 | 6A | j | Lowercase j |
| 107 | 6B | k | Lowercase k |
| 108 | 6C | l | Lowercase l |
| 109 | 6D | m | Lowercase m |
| 110 | 6E | n | Lowercase n |
| 111 | 6F | o | Lowercase o |
| 112 | 70 | p | Lowercase p |
| 113 | 71 | q | Lowercase q |
| 114 | 72 | r | Lowercase r |
| 115 | 73 | s | Lowercase s |
| 116 | 74 | t | Lowercase t |
| 117 | 75 | u | Lowercase u |
| 118 | 76 | v | Lowercase v |
| 119 | 77 | w | Lowercase w |
| 120 | 78 | x | Lowercase x |
| 121 | 79 | y | Lowercase y |
| 122 | 7A | z | Lowercase z |
| 123 | 7B | { | Opening brace |
| 124 | 7C | \| | Vertical line |
| 125 | 7D | } | Closing brace |
| 126 | 7E | ~ | Tilde |
| 127 | 7F | DEL | Delete |

Table C-2. OSV$ASCII6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 21 | ! | Exclamation point |
| 02 | 22 | " | Quotation marks |
| 03 | 23 | # | Number sign |
| 04 | 24 | $ | Dollar sign |
| 05 | 25 | % | Percent sign |
| 06 | 26 | & | Ampersand |
| 07 | 27 | ' | Apostrophe |
| 08 | 28 | ( | Opening parenthesis |
| 09 | 29 | ) | Closing parenthesis |
| 10 | 2A | * | Asterisk |
| 11 | 2B | + | Plus |
| 12 | 2C | , | Comma |
| 13 | 2D | - | Hyphen |
| 14 | 2E | . | Period |
| 15 | 2F | / | Slant |
| 16 | 30 | 0 | Zero |
| 17 | 31 | 1 | One |
| 18 | 32 | 2 | Two |
| 19 | 33 | 3 | Three |
| 20 | 34 | 4 | Four |
| 21 | 35 | 5 | Five |
| 22 | 36 | 6 | Six |
| 23 | 37 | 7 | Seven |
| 24 | 38 | 8 | Eight |
| 25 | 39 | 9 | Nine |
| 26 | 3A | : | Colon |
| 27 | 3B | ; | Semicolon |
| 28 | 3C | < | Less than |
| 29 | 3D | = | Equals |
| 30 | 3E | > | Greater than |
| 31 | 3F | ? | Question mark |
| 32 | 40,60 | @,` | Commercial at, grave accent |
| 33 | 41,61 | A,a | Uppercase A, lowercase a |
| 34 | 42,62 | B,b | Uppercase B, lowercase b |
| 35 | 43,63 | C,c | Uppercase C, lowercase c |
| 36 | 44,64 | D,d | Uppercase D, lowercase d |
| 37 | 45,65 | E,e | Uppercase E, lowercase e |
| 38 | 46,66 | F,f | Uppercase F, lowercase f |
| 39 | 47,67 | G,g | Uppercase G, lowercase g |
| 40 | 48,68 | H,h | Uppercase H, lowercase h |
| 41 | 49,69 | I,i | Uppercase I, lowercase i |
| 42 | 4A,6A | J,j | Uppercase J, lowercase j |
| 43 | 4B,6B | K,k | Uppercase K, lowercase k |
| 44 | 4C,6C | L,l | Uppercase L, lowercase l |
| 45 | 4D,6D | M,m | Uppercase M, lowercase m |

(Continued on next page)

Table C-2. OSV$ASCII6_FOLDED Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 4E,6E | N,n | Uppercase N, lowercase n |
| 47 | 4F,6F | O,o | Uppercase O, lowercase o |
| 48 | 50,70 | P,p | Uppercase P, lowercase p |
| 49 | 51,71 | Q,q | Uppercase Q, lowercase q |
| 50 | 52,72 | R,r | Uppercase R, lowercase r |
| 51 | 53,73 | S,s | Uppercase S, lowercase s |
| 52 | 54,74 | T,t | Uppercase T, lowercase t |
| 53 | 55,75 | U,u | Uppercase U, lowercase u |
| 54 | 56,76 | V,v | Uppercase V, lowercase v |
| 55 | 57,77 | W,w | Uppercase W, lowercase w |
| 56 | 58,78 | X,x | Uppercase X, lowercase x |
| 57 | 59,79 | Y,y | Uppercase Y, lowercase y |
| 58 | 5A,7A | Z,z | Uppercase Z, lowercase z |
| 59 | 5B,7B | [,{ | Opening bracket, opening brace |
| 60 | 5C,7C | \,\| | Reverse slant, vertical line |
| 61 | 5D,7D | ],} | Closing bracket, closing brace |
| 62 | 5E,7E | ^,~ | Circumflex, tilde |
| 63 | 5F | _ | Underline |

Table C-3. OSV$ASCII6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 21 | ! | Exclamation point |
| 02 | 22 | " | Quotation marks |
| 03 | 23 | # | Number sign |
| 04 | 24 | $ | Dollar sign |
| 05 | 25 | % | Percent sign |
| 06 | 26 | & | Ampersand |
| 07 | 27 | ' | Apostrophe |
| 08 | 28 | ( | Opening parenthesis |
| 09 | 29 | ) | Closing parenthesis |
| 10 | 2A | * | Asterisk |
| 11 | 2B | + | Plus |
| 12 | 2C | , | Comma |
| 13 | 2D | - | Hyphen |
| 14 | 2E | . | Period |
| 15 | 2F | / | Slant |
| 16 | 30 | 0 | Zero |
| 17 | 31 | 1 | One |
| 18 | 32 | 2 | Two |
| 19 | 33 | 3 | Three |
| 20 | 34 | 4 | Four |
| 21 | 35 | 5 | Five |
| 22 | 36 | 6 | Six |
| 23 | 37 | 7 | Seven |
| 24 | 38 | 8 | Eight |
| 25 | 39 | 9 | Nine |
| 26 | 3A | : | Colon |
| 27 | 3B | ; | Semicolon |
| 28 | 3C | < | Less than |
| 29 | 3D | = | Equals |
| 30 | 3E | > | Greater than |
| 31 | 3F | ? | Question mark |
| 32 | 40 | @ | Commercial at |
| 33 | 41 | A | Uppercase A |
| 34 | 42 | B | Uppercase B |
| 35 | 43 | C | Uppercase C |
| 36 | 44 | D | Uppercase D |
| 37 | 45 | E | Uppercase E |
| 38 | 46 | F | Uppercase F |
| 39 | 47 | G | Uppercase G |
| 40 | 48 | H | Uppercase H |
| 41 | 49 | I | Uppercase I |
| 42 | 4A | J | Uppercase J |
| 43 | 4B | K | Uppercase K |
| 44 | 4C | L | Uppercase L |
| 45 | 4D | M | Uppercase M |

(Continued on next page)

Table C-3. OSV$ASCII6_STRICT Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 4E | N | Uppercase N |
| 47 | 4F | O | Uppercase O |
| 48 | 50 | P | Uppercase P |
| 49 | 51 | Q | Uppercase Q |
| 50 | 52 | R | Uppercase R |
| 51 | 53 | S | Uppercase S |
| 52 | 54 | T | Uppercase T |
| 53 | 55 | U | Uppercase U |
| 54 | 56 | V | Uppercase V |
| 55 | 57 | W | Uppercase W |
| 56 | 58 | X | Uppercase X |
| 57 | 59 | Y | Uppercase Y |
| 58 | 5A | Z | Uppercase Z |
| 59 | 5B | [ | Opening bracket |
| 60 | 5C | \ | Reverse slant |
| 61 | 5D | ] | Closing bracket |
| 62 | 5E | ^ | Circumflex |
| 63 | 5F | _ | Underline |

Table C-4. OSV$COBOL6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 40,60 | @,` | Commercial at, grave accent |
| 02 | 25 | % | Percent sign |
| 03 | 5B,7B | [,{ | Opening bracket, opening brace |
| 04 | 5F | _ | Underline |
| 05 | 23 | # | Number sign |
| 06 | 26 | & | Ampersand |
| 07 | 27 | ' | Apostrophe |
| 08 | 3F | ? | Question mark |
| 09 | 3E | > | Greater than |
| 10 | 5C,7C | \,\| | Reverse slant, vertical line |
| 11 | 5E,7E | ^,~ | Circumflex, tilde |
| 12 | 2E | . | Period |
| 13 | 29 | ) | Closing parenthesis |
| 14 | 3B | ; | Semicolon |
| 15 | 2B | + | Plus |
| 16 | 24 | $ | Dollar sign |
| 17 | 2A | * | Asterisk |
| 18 | 2D | - | Hyphen |
| 19 | 2F | / | Slant |
| 20 | 2C | , | Comma |
| 21 | 28 | ( | Opening parenthesis |
| 22 | 3D | = | Equals |
| 23 | 22 | " | Quotation marks |
| 24 | 3C | < | Less than |
| 25 | 41,61 | A,a | Uppercase A, lowercase a |
| 26 | 42,62 | B,b | Uppercase B, lowercase b |
| 27 | 43,63 | C,c | Uppercase C, lowercase c |
| 28 | 44,64 | D,d | Uppercase D, lowercase d |
| 29 | 45,65 | E,e | Uppercase E, lowercase e |
| 30 | 46,66 | F,f | Uppercase F, lowercase f |
| 31 | 47,67 | G,g | Uppercase G, lowercase g |
| 32 | 48,68 | H,h | Uppercase H, lowercase h |
| 33 | 49,69 | I,i | Uppercase I, lowercase i |
| 34 | 21 | ! | Exclamation point |
| 35 | 4A,6A | J,j | Uppercase J, lowercase j |
| 36 | 4B,6B | K,k | Uppercase K, lowercase k |
| 37 | 4C,6C | L,l | Uppercase L, lowercase l |
| 38 | 4D,6D | M,m | Uppercase M, lowercase m |
| 39 | 4E,6E | N,n | Uppercase N, lowercase n |
| 40 | 4F,6F | O,o | Uppercase O, lowercase o |
| 41 | 50,70 | P,p | Uppercase P, lowercase p |
| 42 | 51,71 | Q,q | Uppercase Q, lowercase q |
| 43 | 52,72 | R,r | Uppercase R, lowercase r |
| 44 | 5D,7D | ],} | Closing bracket, closing brace |
| 45 | 53,73 | S,s | Uppercase S, lowercase s |

(Continued on next page)

Table C-4. OSV$COBOL6_FOLDED Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 54,74 | T,t | Uppercase T, lowercase t |
| 47 | 55,75 | U,u | Uppercase U, lowercase u |
| 48 | 56,76 | V,v | Uppercase V, lowercase v |
| 49 | 57,77 | W,w | Uppercase W, lowercase w |
| 50 | 58,78 | X,x | Uppercase X, lowercase x |
| 51 | 59,79 | Y,y | Uppercase Y, lowercase y |
| 52 | 5A,7A | Z,z | Uppercase Z, lowercase z |
| 53 | 3A | : | Colon |
| 54 | 30 | 0 | Zero |
| 55 | 31 | 1 | One |
| 56 | 32 | 2 | Two |
| 57 | 33 | 3 | Three |
| 58 | 34 | 4 | Four |
| 59 | 35 | 5 | Five |
| 60 | 36 | 6 | Six |
| 61 | 37 | 7 | Seven |
| 62 | 38 | 8 | Eight |
| 63 | 39 | 9 | Nine |

Table C-5. OSV$COBOL6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 40 | @ | Commercial at |
| 02 | 25 | % | Percent sign |
| 03 | 5B | [ | Opening bracket |
| 04 | 5F | _ | Underline |
| 05 | 23 | # | Number sign |
| 06 | 26 | & | Ampersand |
| 07 | 27 | ' | Apostrophe |
| 08 | 3F | ? | Question mark |
| 09 | 3E | > | Greater than |
| 10 | 5C | \ | Reverse slant |
| 11 | 5E | ^ | Circumflex |
| 12 | 2E | . | Period |
| 13 | 29 | ) | Closing parenthesis |
| 14 | 3B | ; | Semicolon |
| 15 | 2B | + | Plus |
| 16 | 24 | $ | Dollar sign |
| 17 | 2A | * | Asterisk |
| 18 | 2D | - | Hyphen |
| 19 | 2F | / | Slant |
| 20 | 2C | , | Comma |
| 21 | 28 | ( | Opening parenthesis |
| 22 | 3D | = | Equals |
| 23 | 22 | " | Quotation marks |
| 24 | 3C | < | Less than |
| 25 | 41 | A | Uppercase A |
| 26 | 42 | B | Uppercase B |
| 27 | 43 | C | Uppercase C |
| 28 | 44 | D | Uppercase D |
| 29 | 45 | E | Uppercase E |
| 30 | 46 | F | Uppercase F |
| 31 | 47 | G | Uppercase G |
| 32 | 48 | H | Uppercase H |
| 33 | 49 | I | Uppercase I |
| 34 | 21 | ! | Exclamation point |
| 35 | 4A | J | Uppercase J |
| 36 | 4B | K | Uppercase K |
| 37 | 4C | L | Uppercase L |
| 38 | 4D | M | Uppercase M |
| 39 | 4E | N | Uppercase N |
| 40 | 4F | O | Uppercase O |
| 41 | 50 | P | Uppercase P |
| 42 | 51 | Q | Uppercase Q |
| 43 | 52 | R | Uppercase R |
| 44 | 5D | ] | Closing bracket |
| 45 | 53 | S | Uppercase S |

(Continued on next page)

Table C-5. OSV$COBOL6_STRICT Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 54 | T | Uppercase T |
| 47 | 55 | U | Uppercase U |
| 48 | 56 | V | Uppercase V |
| 49 | 57 | W | Uppercase W |
| 50 | 58 | X | Uppercase X |
| 51 | 59 | Y | Uppercase Y |
| 52 | 5A | Z | Uppercase Z |
| 53 | 3A | : | Colon |
| 54 | 30 | 0 | Zero |
| 55 | 31 | 1 | One |
| 56 | 32 | 2 | Two |
| 57 | 33 | 3 | Three |
| 58 | 34 | 4 | Four |
| 59 | 35 | 5 | Five |
| 60 | 36 | 6 | Six |
| 61 | 37 | 7 | Seven |
| 62 | 38 | 8 | Eight |
| 63 | 39 | 9 | Nine |

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 41,61 | A,a | Uppercase A, lowercase a |
| 01 | 42,62 | B,b | Uppercase B, lowercase b |
| 02 | 43,63 | C,c | Uppercase C, lowercase c |
| 03 | 44,64 | D,d | Uppercase D, lowercase d |
| 04 | 45,65 | E,e | Uppercase E, lowercase e |
| 05 | 46,66 | F,f | Uppercase F, lowercase f |
| 06 | 47,67 | G,g | Uppercase G, lowercase g |
| 07 | 48,68 | H,h | Uppercase H, lowercase h |
| 08 | 49,69 | I,i | Uppercase I, lowercase i |
| 09 | 4A,6A | J,j | Uppercase J, lowercase j |
| 10 | 4B,6B | K,k | Uppercase K, lowercase k |
| 11 | 4C,6C | L,l | Uppercase L, lowercase l |
| 12 | 4D,6D | M,m | Uppercase M, lowercase m |
| 13 | 4E,6E | N,n | Uppercase N, lowercase n |
| 14 | 4F,6F | O,o | Uppercase O, lowercase o |
| 15 | 50,70 | P,p | Uppercase P, lowercase p |
| 16 | 51,71 | Q,q | Uppercase Q, lowercase q |
| 17 | 52,72 | R,r | Uppercase R, lowercase r |
| 18 | 53,73 | S,s | Uppercase S, lowercase s |
| 19 | 54,74 | T,t | Uppercase T, lowercase t |
| 20 | 55,75 | U,u | Uppercase U, lowercase u |
| 21 | 56,76 | V,v | Uppercase V, lowercase v |
| 22 | 57,77 | W,w | Uppercase W, lowercase w |
| 23 | 58,78 | X,x | Uppercase X, lowercase x |
| 24 | 59,79 | Y,y | Uppercase Y, lowercase y |
| 25 | 5A,7A | Z,z | Uppercase Z, lowercase z |
| 26 | 30 | 0 | Zero |
| 27 | 31 | 1 | One |
| 28 | 32 | 2 | Two |
| 29 | 33 | 3 | Three |
| 30 | 34 | 4 | Four |
| 31 | 35 | 5 | Five |
| 32 | 36 | 6 | Six |
| 33 | 37 | 7 | Seven |
| 34 | 38 | 8 | Eight |
| 35 | 39 | 9 | Nine |
| 36 | 2B | + | Plus |
| 37 | 2D | - | Hyphen |
| 38 | 2A | * | Asterisk |
| 39 | 2F | / | Slant |
| 40 | 28 | ( | Opening parenthesis |
| 41 | 29 | ) | Closing parenthesis |
| 42 | 24 | $ | Dollar sign |
| 43 | 3D | = | Equals |
| 44 | 20 | SP | Space |
| 45 | 2C | , | Comma |

(Continued on next page)

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 2E | . | Period |
| 47 | 23 | # | Number sign |
| 48 | 5B,7B | [,{ | Opening bracket, opening brace |
| 49 | 5D,7D | ],} | Closing bracket, closing brace |
| 50 | 3A | : | Colon |
| 51 | 22 | " | Quotation marks |
| 52 | 5F | _ | Underline |
| 53 | 21 | ! | Exclamation point |
| 54 | 26 | & | Ampersand |
| 55 | 27 | ' | Apostrophe |
| 56 | 3F | ? | Question mark |
| 57 | 3C | < | Less than |
| 58 | 3E | > | Greater than |
| 59 | 40,60 | @,` | Commercial at, grave accent |
| 60 | 5C,7C | \,\| | Reverse slant, vertical line |
| 61 | 5E,7E | ^,~ | Circumflex, tilde |
| 62 | 3B | ; | Semicolon |

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 41 | A | Uppercase A |
| 01 | 42 | B | Uppercase B |
| 02 | 43 | C | Uppercase C |
| 03 | 44 | D | Uppercase D |
| 04 | 45 | E | Uppercase E |
| 05 | 46 | F | Uppercase F |
| 06 | 47 | G | Uppercase G |
| 07 | 48 | H | Uppercase H |
| 08 | 49 | I | Uppercase I |
| 09 | 4A | J | Uppercase J |
| 10 | 4B | K | Uppercase K |
| 11 | 4C | L | Uppercase L |
| 12 | 4D | M | Uppercase M |
| 13 | 4E | N | Uppercase N |
| 14 | 4F | O | Uppercase O |
| 15 | 50 | P | Uppercase P |
| 16 | 51 | Q | Uppercase Q |
| 17 | 52 | R | Uppercase R |
| 18 | 53 | S | Uppercase S |
| 19 | 54 | T | Uppercase T |
| 20 | 55 | U | Uppercase U |
| 21 | 56 | V | Uppercase V |
| 22 | 57 | W | Uppercase W |
| 23 | 58 | X | Uppercase X |
| 24 | 59 | Y | Uppercase Y |
| 25 | 5A | Z | Uppercase Z |
| 26 | 30 | 0 | Zero |
| 27 | 31 | 1 | One |
| 28 | 32 | 2 | Two |
| 29 | 33 | 3 | Three |
| 30 | 34 | 4 | Four |
| 31 | 35 | 5 | Five |
| 32 | 36 | 6 | Six |
| 33 | 37 | 7 | Seven |
| 34 | 38 | 8 | Eight |
| 35 | 39 | 9 | Nine |
| 36 | 2B | + | Plus |
| 37 | 2D | - | Hyphen |
| 38 | 2A | * | Asterisk |
| 39 | 2F | / | Slant |
| 40 | 28 | ( | Opening parenthesis |
| 41 | 29 | ) | Closing parenthesis |
| 42 | 24 | $ | Dollar sign |
| 43 | 3D | = | Equals |
| 44 | 20 | SP | Space |
| 45 | 2C | , | Comma |

(Continued on next page)

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 2E | . | Period |
| 47 | 23 | # | Number sign |
| 48 | 5B | [ | Opening bracket |
| 49 | 5D | ] | Closing bracket |
| 50 | 3A | : | Colon |
| 51 | 22 | " | Quotation marks |
| 52 | 5F | _ | Underline |
| 53 | 21 | ! | Exclamation point |
| 54 | 26 | & | Ampersand |
| 55 | 27 | ' | Apostrophe |
| 56 | 3F | ? | Question mark |
| 57 | 3C | < | Less than |
| 58 | 3E | > | Greater than |
| 59 | 40 | @ | Commercial at |
| 60 | 5C | \ | Reverse slant |
| 61 | 5E | ^ | Circumflex |
| 62 | 3B | ; | Semicolon |

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 3A | : | Colon |
| 01 | 41,61 | A,a | Uppercase A, lowercase a |
| 02 | 42,62 | B,b | Uppercase B, lowercase b |
| 03 | 43,63 | C,c | Uppercase C, lowercase c |
| 04 | 44,64 | D,d | Uppercase D, lowercase d |
| 05 | 45,65 | E,e | Uppercase E, lowercase e |
| 06 | 46,66 | F,f | Uppercase F, lowercase f |
| 07 | 47,67 | G,g | Uppercase G, lowercase g |
| 08 | 48,68 | H,h | Uppercase H, lowercase h |
| 09 | 49,69 | I,i | Uppercase I, lowercase i |
| 10 | 4A,6A | J,j | Uppercase J, lowercase j |
| 11 | 4B,6B | K,k | Uppercase K, lowercase k |
| 12 | 4C,6C | L,l | Uppercase L, lowercase l |
| 13 | 4D,6D | M,m | Uppercase M, lowercase m |
| 14 | 4E,6E | N,n | Uppercase N, lowercase n |
| 15 | 4F,6F | O,o | Uppercase O, lowercase o |
| 16 | 50,70 | P,p | Uppercase P, lowercase p |
| 17 | 51,71 | Q,q | Uppercase Q, lowercase q |
| 18 | 52,72 | R,r | Uppercase R, lowercase r |
| 19 | 53,73 | S,s | Uppercase S, lowercase s |
| 20 | 54,74 | T,t | Uppercase T, lowercase t |
| 21 | 55,75 | U,u | Uppercase U, lowercase u |
| 22 | 56,76 | V,v | Uppercase V, lowercase v |
| 23 | 57,77 | W,w | Uppercase W, lowercase w |
| 24 | 58,78 | X,x | Uppercase X, lowercase x |
| 25 | 59,79 | Y,y | Uppercase Y, lowercase y |
| 26 | 5A,7A | Z,z | Uppercase Z, lowercase z |
| 27 | 30 | 0 | Zero |
| 28 | 31 | 1 | One |
| 29 | 32 | 2 | Two |
| 30 | 33 | 3 | Three |
| 31 | 34 | 4 | Four |
| 32 | 35 | 5 | Five |
| 33 | 36 | 6 | Six |
| 34 | 37 | 7 | Seven |
| 35 | 38 | 8 | Eight |
| 36 | 39 | 9 | Nine |
| 37 | 2B | + | Plus |
| 38 | 2D | - | Hyphen |
| 39 | 2A | * | Asterisk |
| 40 | 2F | / | Slant |
| 41 | 28 | ( | Opening parenthesis |
| 42 | 29 | ) | Closing parenthesis |
| 43 | 24 | $ | Dollar sign |
| 44 | 3D | = | Equals |
| 45 | 20 | SP | Space |

(Continued on next page)

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 2C | , | Comma |
| 47 | 2E | . | Period |
| 48 | 23 | # | Number sign |
| 49 | 5B,7B | [,{ | Opening bracket, opening brace |
| 50 | 5D,7D | ],} | Closing bracket, closing brace |
| 51 | 25 | % | Percent sign |
| 52 | 22 | " | Quotation marks |
| 53 | 5F | _ | Underline |
| 54 | 21 | ! | Exclamation point |
| 55 | 26 | & | Ampersand |
| 56 | 27 | ' | Apostrophe |
| 57 | 3F | ? | Question mark |
| 58 | 3C | < | Less than |
| 59 | 3E | > | Greater than |
| 60 | 40,60 | @,` | Commercial at, grave accent |
| 61 | 5C,7C | \,\| | Reverse slant, vertical line |
| 62 | 5E,7E | ^,~ | Circumflex, tilde |
| 63 | 3B | ; | Semicolon |

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 3A | : | Colon |
| 01 | 41 | A | Uppercase A |
| 02 | 42 | B | Uppercase B |
| 03 | 43 | C | Uppercase C |
| 04 | 44 | D | Uppercase D |
| 05 | 45 | E | Uppercase E |
| 06 | 46 | F | Uppercase F |
| 07 | 47 | G | Uppercase G |
| 08 | 48 | H | Uppercase H |
| 09 | 49 | I | Uppercase I |
| 10 | 4A | J | Uppercase J |
| 11 | 4B | K | Uppercase K |
| 12 | 4C | L | Uppercase L |
| 13 | 4D | M | Uppercase M |
| 14 | 4E | N | Uppercase N |
| 15 | 4F | O | Uppercase O |
| 16 | 50 | P | Uppercase P |
| 17 | 51 | Q | Uppercase Q |
| 18 | 52 | R | Uppercase R |
| 19 | 53 | S | Uppercase S |
| 20 | 54 | T | Uppercase T |
| 21 | 55 | U | Uppercase U |
| 22 | 56 | V | Uppercase V |
| 23 | 57 | W | Uppercase W |
| 24 | 58 | X | Uppercase X |
| 25 | 59 | Y | Uppercase Y |
| 26 | 5A | Z | Uppercase Z |
| 27 | 30 | 0 | Zero |
| 28 | 31 | 1 | One |
| 29 | 32 | 2 | Two |
| 30 | 33 | 3 | Three |
| 31 | 34 | 4 | Four |
| 32 | 35 | 5 | Five |
| 33 | 36 | 6 | Six |
| 34 | 37 | 7 | Seven |
| 35 | 38 | 8 | Eight |
| 36 | 39 | 9 | Nine |
| 37 | 2B | + | Plus |
| 38 | 2D | - | Hyphen |
| 39 | 2A | * | Asterisk |
| 40 | 2F | / | Slant |
| 41 | 28 | ( | Opening parenthesis |
| 42 | 29 | ) | Closing parenthesis |
| 43 | 24 | $ | Dollar sign |
| 44 | 3D | = | Equals |
| 45 | 20 | SP | Space |

(Continued on next page)

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 2C | , | Comma |
| 47 | 2E | . | Period |
| 48 | 23 | # | Number sign |
| 49 | 5B | [ | Opening bracket |
| 50 | 5D | ] | Closing bracket |
| 51 | 25 | % | Percent sign |
| 52 | 22 | " | Quotation marks |
| 53 | 5F | _ | Underline |
| 54 | 21 | ! | Exclamation point |
| 55 | 26 | & | Ampersand |
| 56 | 27 | ' | Apostrophe |
| 57 | 3F | ? | Question mark |
| 58 | 3C | < | Less than |
| 59 | 3E | > | Greater than |
| 60 | 40 | @ | Commercial at |
| 61 | 5C | \ | Reverse slant |
| 62 | 5E | ^ | Circumflex |
| 63 | 3B | ; | Semicolon |

Table C-10. OSV$EBCDIC Collating Sequence

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 000 | 00 | NUL | Null |
| 001 | 01 | SOH | Start of heading |
| 002 | 02 | STX | Start of text |
| 003 | 03 | ETX | End of text |
| 004 | 9C | --- | Unassigned |
| 005 | 09 | HT | Horizontal tabulation |
| 006 | 86 | --- | Unassigned |
| 007 | 7F | DEL | Delete |
| 008 | 97 | --- | Unassigned |
| 009 | 8D | --- | Unassigned |
| 010 | 8E | --- | Unassigned |
| 011 | 0B | VT | Vertical tabulation |
| 012 | 0C | FF | Form feed |
| 013 | 0D | CR | Carriage return |
| 014 | 0E | SO | Shift out |
| 015 | 0F | SI | Shift in |
| 016 | 10 | DLE | Data link escape |
| 017 | 11 | DC1 | Device control 1 |
| 018 | 12 | DC2 | Device control 2 |
| 019 | 13 | DC3 | Device control 3 |
| 020 | 9D | --- | Unassigned |
| 021 | 85 | --- | Unassigned |
| 022 | 08 | BS | Backspace |
| 023 | 87 | --- | Unassigned |
| 024 | 18 | CAN | Cancel |
| 025 | 19 | EM | End of medium |
| 026 | 92 | --- | Unassigned |
| 027 | 8F | --- | Unassigned |
| 028 | 1C | FS | File separator |
| 029 | 1D | GS | Group separator |
| 030 | 1E | RS | Record separator |
| 031 | 1F | US | Unit separator |
| 032 | 80 | --- | Unassigned |
| 033 | 81 | --- | Unassigned |
| 034 | 82 | --- | Unassigned |
| 035 | 83 | --- | Unassigned |
| 036 | 84 | --- | Unassigned |
| 037 | 0A | LF | Line feed |
| 038 | 17 | ETB | End of transmission block |
| 039 | 1B | ESC | Escape |
| 040 | 88 | --- | Unassigned |
| 041 | 89 | --- | Unassigned |
| 042 | 8A | --- | Unassigned |
| 043 | 8B | --- | Unassigned |
| 044 | 8C | --- | Unassigned |
| 045 | 05 | ENQ | Enquiry |
| 046 | 06 | ACK | Acknowledge |
| 047 | 07 | BEL | Bell |
| 048 | 90 | --- | Unassigned |
| 049 | 91 | --- | Unassigned |
| 050 | 16 | SYN | Synchronous idle |
| 051 | 93 | --- | Unassigned |
| 052 | 94 | --- | Unassigned |

(Continued on next page)

Table C-10. OSV$EBCDIC Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 053 | 95 | --- | Unassigned |
| 054 | 96 | --- | Unassigned |
| 055 | 04 | EOT | End of transmission |
| 056 | 98 | --- | Unassigned |
| 057 | 99 | --- | Unassigned |
| 058 | 9A | --- | Unassigned |
| 059 | 9B | --- | Unassigned |
| 060 | 14 | DC4 | Device control 4 |
| 061 | 15 | NAK | Negative acknowledge |
| 062 | 9E | --- | Unassigned |
| 063 | 1A | SUB | Substitute |
| 064 | 20 | SP | Space |
| 065 | A0 | --- | Unassigned |
| 066 | A1 | --- | Unassigned |
| 067 | A2 | --- | Unassigned |
| 068 | A3 | --- | Unassigned |
| 069 | A4 | --- | Unassigned |
| 070 | A5 | --- | Unassigned |
| 071 | A6 | --- | Unassigned |
| 072 | A7 | --- | Unassigned |
| 073 | A8 | --- | Unassigned |
| 074 | 5B | [ | Opening bracket |
| 075 | 2E | . | Period |
| 076 | 3C | < | Less than |
| 077 | 28 | ( | Opening parenthesis |
| 078 | 2B | + | Plus |
| 079 | 21 | ! | Exclamation point |
| 080 | 26 | & | Ampersand |
| 081 | A9 | --- | Unassigned |
| 082 | AA | --- | Unassigned |
| 083 | AB | --- | Unassigned |
| 084 | AC | --- | Unassigned |
| 085 | AD | --- | Unassigned |
| 086 | AE | --- | Unassigned |
| 087 | AF | --- | Unassigned |
| 088 | B0 | --- | Unassigned |
| 089 | B1 | --- | Unassigned |
| 090 | 5D | ] | Closing bracket |
| 091 | 24 | $ | Dollar sign |
| 092 | 2A | * | Asterisk |
| 093 | 29 | ) | Closing parenthesis |
| 094 | 3B | ; | Semicolon |
| 095 | 5E | ^ | Circumflex |
| 096 | 2D | - | Hyphen |
| 097 | 2F | / | Slant |
| 098 | B2 | --- | Unassigned |
| 099 | B3 | --- | Unassigned |
| 100 | B4 | --- | Unassigned |
| 101 | B5 | --- | Unassigned |
| 102 | B6 | --- | Unassigned |
| 103 | B7 | --- | Unassigned |
| 104 | B8 | --- | Unassigned |
| 105 | B9 | --- | Unassigned |

(Continued on next page)

Table C-10. OSV$EBCDIC Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 106 | 7C | \| | Vertical line |
| 107 | 2C | , | Comma |
| 108 | 25 | % | Percent sign |
| 109 | 5F | _ | Underline |
| 110 | 3E | > | Greater than |
| 111 | 3F | ? | Question mark |
| 112 | BA | --- | Unassigned |
| 113 | BB | --- | Unassigned |
| 114 | BC | --- | Unassigned |
| 115 | BD | --- | Unassigned |
| 116 | BE | --- | Unassigned |
| 117 | BF | --- | Unassigned |
| 118 | C0 | --- | Unassigned |
| 119 | C1 | --- | Unassigned |
| 120 | C2 | --- | Unassigned |
| 121 | 60 | ` | Grave accent |
| 122 | 3A | : | Colon |
| 123 | 23 | # | Number sign |
| 124 | 40 | @ | Commercial at |
| 125 | 27 | ' | Apostrophe |
| 126 | 3D | = | Equals |
| 127 | 22 | " | Quotation marks |
| 128 | C3 | --- | Unassigned |
| 129 | 61 | a | Lowercase a |
| 130 | 62 | b | Lowercase b |
| 131 | 63 | c | Lowercase c |
| 132 | 64 | d | Lowercase d |
| 133 | 65 | e | Lowercase e |
| 134 | 66 | f | Lowercase f |
| 135 | 67 | g | Lowercase g |
| 136 | 68 | h | Lowercase h |
| 137 | 69 | i | Lowercase i |
| 138 | C4 | --- | Unassigned |
| 139 | C5 | --- | Unassigned |
| 140 | C6 | --- | Unassigned |
| 141 | C7 | --- | Unassigned |
| 142 | C8 | --- | Unassigned |
| 143 | C9 | --- | Unassigned |
| 144 | CA | --- | Unassigned |
| 145 | 6A | j | Lowercase j |
| 146 | 6B | k | Lowercase k |
| 147 | 6C | l | Lowercase l |
| 148 | 6D | m | Lowercase m |
| 149 | 6E | n | Lowercase n |
| 150 | 6F | o | Lowercase o |
| 151 | 70 | p | Lowercase p |
| 152 | 71 | q | Lowercase q |
| 153 | 72 | r | Lowercase r |
| 154 | CB | --- | Unassigned |
| 155 | CC | --- | Unassigned |
| 156 | CD | --- | Unassigned |
| 157 | CE | --- | Unassigned |
| 158 | CF | --- | Unassigned |

(Continued on next page)

Table C-10. OSV$EBCDIC Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 159 | D0 | --- | Unassigned |
| 160 | D1 | --- | Unassigned |
| 161 | 7E | --- | Unassigned |
| 162 | 73 | s | Lowercase s |
| 163 | 74 | t | Lowercase t |
| 164 | 75 | u | Lowercase u |
| 165 | 76 | v | Lowercase v |
| 166 | 77 | w | Lowercase w |
| 167 | 78 | x | Lowercase x |
| 168 | 79 | y | Lowercase y |
| 169 | 7A | z | Lowercase z |
| 170 | D2 | --- | Unassigned |
| 171 | D3 | --- | Unassigned |
| 172 | D4 | --- | Unassigned |
| 173 | D5 | --- | Unassigned |
| 174 | D6 | --- | Unassigned |
| 175 | D7 | --- | Unassigned |
| 176 | D8 | --- | Unassigned |
| 177 | D9 | --- | Unassigned |
| 178 | DA | --- | Unassigned |
| 179 | DB | --- | Unassigned |
| 180 | DC | --- | Unassigned |
| 181 | DD | --- | Unassigned |
| 182 | DE | --- | Unassigned |
| 183 | DF | --- | Unassigned |
| 184 | E0 | --- | Unassigned |
| 185 | E1 | --- | Unassigned |
| 186 | E2 | --- | Unassigned |
| 187 | E3 | --- | Unassigned |
| 188 | E4 | --- | Unassigned |
| 189 | E5 | --- | Unassigned |
| 190 | E6 | --- | Unassigned |
| 191 | E7 | --- | Unassigned |
| 192 | 7B | { | Opening brace |
| 193 | 41 | A | Uppercase A |
| 194 | 42 | B | Uppercase B |
| 195 | 43 | C | Uppercase C |
| 196 | 44 | D | Uppercase D |
| 197 | 45 | E | Uppercase E |
| 198 | 46 | F | Uppercase F |
| 199 | 47 | G | Uppercase G |
| 200 | 48 | H | Uppercase H |
| 201 | 49 | I | Uppercase I |
| 202 | E8 | --- | Unassigned |
| 203 | E9 | --- | Unassigned |
| 204 | EA | --- | Unassigned |
| 205 | EB | --- | Unassigned |
| 206 | EC | --- | Unassigned |
| 207 | ED | --- | Unassigned |
| 208 | 7D | } | Closing brace |
| 209 | 4A | J | Uppercase J |
| 210 | 4B | K | Uppercase K |
| 211 | 4C | L | Uppercase L |

(Continued on next page)

Table C-10. OSV$EBCDIC Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 212 | 4D | M | Uppercase M |
| 213 | 4E | N | Uppercase N |
| 214 | 4F | O | Uppercase O |
| 215 | 50 | P | Uppercase P |
| 216 | 51 | Q | Uppercase Q |
| 217 | 52 | R | Uppercase R |
| 218 | EE | --- | Unassigned |
| 219 | EF | --- | Unassigned |
| 220 | F0 | --- | Unassigned |
| 221 | F1 | --- | Unassigned |
| 222 | F2 | --- | Unassigned |
| 223 | F3 | --- | Unassigned |
| 224 | 5C | \ | Reverse slant |
| 225 | 9F | --- | Unassigned |
| 226 | 53 | S | Uppercase S |
| 227 | 54 | T | Uppercase T |
| 228 | 55 | U | Uppercase U |
| 229 | 56 | V | Uppercase V |
| 230 | 57 | W | Uppercase W |
| 231 | 58 | X | Uppercase X |
| 232 | 59 | Y | Uppercase Y |
| 233 | 5A | Z | Uppercase Z |
| 234 | F4 | --- | Unassigned |
| 235 | F5 | --- | Unassigned |
| 236 | F6 | --- | Unassigned |
| 237 | F7 | --- | Unassigned |
| 238 | F8 | --- | Unassigned |
| 239 | F9 | --- | Unassigned |
| 240 | 30 | 0 | Zero |
| 241 | 31 | 1 | One |
| 242 | 32 | 2 | Two |
| 243 | 33 | 3 | Three |
| 244 | 34 | 4 | Four |
| 245 | 35 | 5 | Five |
| 246 | 36 | 6 | Six |
| 247 | 37 | 7 | Seven |
| 248 | 38 | 8 | Eight |
| 249 | 39 | 9 | Nine |
| 250 | FA | --- | Unassigned |
| 251 | FB | --- | Unassigned |
| 252 | FC | --- | Unassigned |
| 253 | FD | --- | Unassigned |
| 254 | FE | --- | Unassigned |
| 255 | FF | --- | Unassigned |

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 2E | . | Period |
| 02 | 3C | < | Less than |
| 03 | 28 | ( | Opening parenthesis |
| 04 | 2B | + | Plus |
| 05 | 21 | ! | Exclamation point |
| 06 | 26 | & | Ampersand |
| 07 | 24 | $ | Dollar sign |
| 08 | 2A | * | Asterisk |
| 09 | 29 | ) | Closing parenthesis |
| 10 | 3B | ; | Semicolon |
| 11 | 5E,7E | ^,~ | Circumflex, tilde |
| 12 | 2D | - | Hyphen |
| 13 | 2F | / | Slant |
| 14 | 2C | , | Comma |
| 15 | 25 | % | Percent sign |
| 16 | 5F | _ | Underline |
| 17 | 3E | > | Greater than |
| 18 | 3F | ? | Question mark |
| 19 | 3A | : | Colon |
| 20 | 23 | # | Number sign |
| 21 | 40,60 | @,` | Commercial at, grave accent |
| 22 | 27 | ' | Apostrophe |
| 23 | 3D | = | Equals |
| 24 | 22 | " | Quotation marks |
| 25 | 5B,7B | [,{ | Opening bracket, opening brace |
| 26 | 41,61 | A,a | Uppercase A, lowercase a |
| 27 | 42,62 | B,b | Uppercase B, lowercase b |
| 28 | 43,63 | C,c | Uppercase C, lowercase c |
| 29 | 44,64 | D,d | Uppercase D, lowercase d |
| 30 | 45,65 | E,e | Uppercase E, lowercase e |
| 31 | 46,66 | F,f | Uppercase F, lowercase f |
| 32 | 47,67 | G,g | Uppercase G, lowercase g |
| 33 | 48,68 | H,h | Uppercase H, lowercase h |
| 34 | 49,69 | I,i | Uppercase I, lowercase i |
| 35 | 5D,7D | ],} | Closing bracket, closing brace |
| 36 | 4A,6A | J,j | Uppercase J, lowercase j |
| 37 | 4B,6B | K,k | Uppercase K, lowercase k |
| 38 | 4C,6C | L,l | Uppercase L, lowercase l |
| 39 | 4D,6D | M,m | Uppercase M, lowercase m |
| 40 | 4E,6E | N,n | Uppercase N, lowercase n |
| 41 | 4F,6F | O,o | Uppercase O, lowercase o |
| 42 | 50,70 | P,p | Uppercase P, lowercase p |
| 43 | 51,71 | Q,q | Uppercase Q, lowercase q |
| 44 | 52,72 | R,r | Uppercase R, lowercase r |
| 45 | 5C,7C | \,\| | Reverse slant, vertical line |

(Continued on next page)

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 53,73 | S,s | Uppercase S, lowercase s |
| 47 | 54,74 | T,t | Uppercase T, lowercase t |
| 48 | 55,75 | U,u | Uppercase U, lowercase u |
| 49 | 56,76 | V,v | Uppercase V, lowercase v |
| 50 | 57,77 | W,w | Uppercase W, lowercase w |
| 51 | 58,78 | X,x | Uppercase X, lowercase x |
| 52 | 59,79 | Y,y | Uppercase Y, lowercase y |
| 53 | 5A,7A | Z,z | Uppercase Z, lowercase z |
| 54 | 30 | 0 | Zero |
| 55 | 31 | 1 | One |
| 56 | 32 | 2 | Two |
| 57 | 33 | 3 | Three |
| 58 | 34 | 4 | Four |
| 59 | 35 | 5 | Five |
| 60 | 36 | 6 | Six |
| 61 | 37 | 7 | Seven |
| 62 | 38 | 8 | Eight |
| 63 | 39 | 9 | Nine |

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 00 | 20 | SP | Space |
| 01 | 2E | . | Period |
| 02 | 3C | < | Less than |
| 03 | 28 | ( | Opening parenthesis |
| 04 | 2B | + | Plus |
| 05 | 21 | ! | Exclamation point |
| 06 | 26 | & | Ampersand |
| 07 | 24 | $ | Dollar sign |
| 08 | 2A | * | Asterisk |
| 09 | 29 | ) | Closing parenthesis |
| 10 | 3B | ; | Semicolon |
| 11 | 5E | ^ | Circumflex |
| 12 | 2D | - | Hyphen |
| 13 | 2F | / | Slant |
| 14 | 2C | , | Comma |
| 15 | 25 | % | Percent sign |
| 16 | 5F | _ | Underline |
| 17 | 3E | > | Greater than |
| 18 | 3F | ? | Question mark |
| 19 | 3A | : | Colon |
| 20 | 23 | # | Number sign |
| 21 | 40 | @ | Commercial at |
| 22 | 27 | ' | Apostrophe |
| 23 | 3D | = | Equals |
| 24 | 22 | " | Quotation marks |
| 25 | 5B | [ | Opening bracket |
| 26 | 41 | A | Uppercase A |
| 27 | 42 | B | Uppercase B |
| 28 | 43 | C | Uppercase C |
| 29 | 44 | D | Uppercase D |
| 30 | 45 | E | Uppercase E |
| 31 | 46 | F | Uppercase F |
| 32 | 47 | G | Uppercase G |
| 33 | 48 | H | Uppercase H |
| 34 | 49 | I | Uppercase I |
| 35 | 5D | ] | Closing bracket |
| 36 | 4A | J | Uppercase J |
| 37 | 4B | K | Uppercase K |
| 38 | 4C | L | Uppercase L |
| 39 | 4D | M | Uppercase M |
| 40 | 4E | N | Uppercase N |
| 41 | 4F | O | Uppercase O |
| 42 | 50 | P | Uppercase P |
| 43 | 51 | Q | Uppercase Q |
| 44 | 52 | R | Uppercase R |
| 45 | 5C | \ | Reverse slant |

(Continued on next page)

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence

(Continued from previous page)

| Collating Sequence Position | ASCII Code (Hexadecimal) | Graphic or Mnemonic | Name or Meaning |
|---|---|---|---|
| 46 | 53 | S | Uppercase S |
| 47 | 54 | T | Uppercase T |
| 48 | 55 | U | Uppercase U |
| 49 | 56 | V | Uppercase V |
| 50 | 57 | W | Uppercase W |
| 51 | 58 | X | Uppercase X |
| 52 | 59 | Y | Uppercase Y |
| 53 | 5A | Z | Uppercase Z |
| 54 | 30 | 0 | Zero |
| 55 | 31 | 1 | One |
| 56 | 32 | 2 | Two |
| 57 | 33 | 3 | Three |
| 58 | 34 | 4 | Four |
| 59 | 35 | 5 | Five |
| 60 | 36 | 6 | Six |
| 61 | 37 | 7 | Seven |
| 62 | 38 | 8 | Eight |
| 63 | 39 | 9 | Nine |

# Unsupported ANSI COBOL Features D

The following ANSI features are implemented in COBOL 5 but not in NOS/VE COBOL.

## Communication Facility (MCS)

The Communication Facility is not available on NOS/VE.

## FILE STATUS

The returned status codes are expanded to conform to the proposed new ANSI standard. Old ones are still the same; existing programs should not be affected. The only impact is that codes that resulted in a 90 or 99 status and aborted the job no longer abort the job and now have have another status code.

## Labels

The operating system does not support any user label records on any file. However, the compiler does accept label declarations in the FD entry for a file.

## RERUN Clause

There is no checkpoint-restart facility.

## REWRITE Statement

The REWRITE statement for sequential files is not supported.

## Segmentation

Because there are no overlays on NOS/VE, segmentation is not supported.

The concept of initial state overlays is not valid in NOS/VE COBOL. In a NOS/VE COBOL program, any sections that are numbered 50 or greater could function differently than with COBOL 5. The differences occur in altered GO TO statements (through the ALTER statement) and in PERFORM exits. If all PERFORM statements exit normally and the ALTER statement is not used, no conversion of the program is necessary; otherwise, manual conversion is necessary.

You can use the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph of the Environment Division to change the range designations, but the clause has no effect on the program.

# FORTRAN Default FIT Field Values E

FORTRAN default file information table (FIT) field values are provided for your reference. You may need this information when coding FILE commands for CYBER 170 files used with FORTRAN FMA. The values listed in the tables describe files used with NOS FORTRAN 5 when your application does not describe the files in a FILE command.

When using FMA, however, the same set of default values do not apply. FMA uses CYBER Record Manager default values, not FORTRAN default values. Therefore, you may need to specify field values when executing with FMA that you got by default when executing on NOS. The CYBER 170 default FIT fields are described in the following tables:

E-1 For Formatted, NAMELIST, and List Directed Sequential I/O

E-2 For Unformatted Sequential I/O

E-3 For Direct Access Formatted and Unformatted I/O

Table E-1. NOS FORTRAN Default FIT Fields for Formatted, Namelist, and List Directed Sequential I/O

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| CIO buffer size (words) | BFS† | † |
| Buffer below highest address | BBH | 0 |
| Block type | BT | C |
| Close flag (positioning of file after close) | CF | N |
| Length in characters of record after trailer count field (T type records only) | CL | 0 |
| Conversion mode | CM | YES |
| Beginning character position of trailer count field, numbered from zero (T type records only) | CP | 0 |
| Length field (D type records) or trailer count field (T type records) is binary | CI | NO |
| Type of information to be listed in dayfile | DFC | 3 |
| Type of information to be listed in error file | EFC | 0 |
| Error options | EO | AD |
| Trivial error limit | ERL | 0 |
| Fatal Flush | FF | 0 |
| Length in characters of an F or Z type record (same as MRL) | FL | 150†† |
| File organization | FO | SQ |
| Character length of fixed header for T type records | HL | 0 |
| Length of user's label area (number of characters) | LBL | 0 |
| Logical file name | LFN | ††† |
| Length in characters of record length field (D type records) | LL | 0 |
| Beginning character position of record length, numbered from zero (D type records) | LP | 0 |
| Label type | LT | ANY |
| Maximum block length in characters | MBL | 0 |
| Minimum block length in characters | MNB | 0 |

(Continued on next page)

Table E-1. NOS FORTRAN Default FIT Fields for Formatted, Namelist, and List Directed Sequential I/O

(Continued from previous page)

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| Minimum record length in characters | MNR | 0 |
| Maximum record length in characters | MRL†† | n/a |
| Multiple of characters per K, E type block | MUL | 2 |
| Open flag (positioning of file after open) | OF | N |
| Padding character for K, E type blocks | PC | 76B |
| Processing direction | PD | IO |
| Number of records per K type block | RB | 1 |
| Record mark character (R records) | RMK | 62B |
| Record type | RT | Z |
| Length field (D type records) or trailer count field (T type records) has sign overpunch | SB | NO |
| Suppress buffering | SBF | NO |
| Suppress read ahead | SPR | NO |
| Character length of trailer portion of T type records | TL | 0 |
| User label processing | ULP | NO |
| End of volume flag (positioning of file at volume CLOSEM time) | VF | U |

†For use with FMA, buffer is determined by CYBER Record Manager using the record length.

††For use with FMA, you must specify an FL that equals or exceeds the record length. When executing on CYBER 170, you can change the default on the PROGRAM or OPEN statement. For formatted, NAMELIST, and list directed READ/WRITE statements, a FILE command can decrease but not increase the maximum record length declared on the PROGRAM statement (static mode).

†††For FMA, set by a FILE command and by the EXECUTE_MIGRATION_TASK command, PROGRAM or OPEN statements. For regular use on CYBER 170, set by PROGRAM or OPEN statements, or by an execution control statement or a FILE command.

Table E-2. NOS FORTRAN Default FIT Fields Unformatted Sequential I/O

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| CIO buffer size (words) | BFS† | † |
| Buffer below highest address | BBH | 0 |
| Block type | BT | I |
| Close flag (positioning of file after close) | CF | N |
| Length in characters of record after trailer count field (T type records only) | CL | 0 |
| Conversion mode | CM | NO |
| Beginning character position of trailer count field, numbered from zero (T type records only) | CP | 0 |
| Length field (D type records) or trailer count field (T type records) is binary | C1 | NO |
| Type of information to be listed in dayfile | DFC | 3 |
| Type of information to be listed in error file | EFC | 0 |
| Error options | EO | AD |
| Trivial error limit | ERL | 0 |
| Fatal Flush | FF | 0 |
| Length in characters of an F or Z type record (same as MRL) | FL | n/a |
| File organization | FO | SQ |
| Character length of fixed header for T type records | HL | 0 |
| Length of user's label area (number of characters) | LBL | 0 |
| Logical file name | LFN | †† |
| Length in characters of record length field (D type records) | LL | 0 |
| Beginning character position of record length, numbered from zero (D type records) | LP | 0 |
| Label type | LT | ANY |
| Maximum block length in characters | MBL | 0 |

(Continued on next page)

Table E-2. NOS FORTRAN Default FIT Fields Unformatted Sequential I/O

(Continued from previous page)

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| Minimum block length in characters | MNB | 0 |
| Minimum record length in characters | MNR | 0 |
| Maximum record length in characters | MRL††† | (2**23)-1 |
| Multiple of characters per K, E type block | MUL | 2 |
| Open flag (positioning of file after open) | OF | N |
| Padding character for K, E type blocks | PC | 76B |
| Processing direction | PD | IO |
| Number of records per K type block | RB | 1 |
| Record mark character (R records) | RMK | n/a |
| Record type | RT | W |
| Length field (D type records) or trailer count field (T type records) has sign overpunch | SB | NO |
| Suppress buffering | SBF | NO |
| Suppress read ahead | SPR | NO |
| Character length of trailer portion of T type records | TL | 0 |
| User label processing | ULP | NO |
| End of volume flag (positioning of file at volume CLOSEM time) | VF | U |

†For use with FMA, buffer is determined by CYBER Record Manager using the record length.

††For FMA, set by a FILE command and by the EXECUTE_MIGRATION_TASK command, PROGRAM or OPEN statements. For regular use on CYBER 170, set by PROGRAM or OPEN statements, or by an execution control statement or a FILE command.

†††For use with FMA, you must specify an MRL that equals or exceeds the record length. When running on CYBER 170, you can change the default on the PROGRAM or OPEN statement.

Table E-3. NOS FORTRAN Default FIT Fields for Direct Access Formatted and Unformatted I/O

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| CIO buffer size (words) | BFS† | † |
| Buffer below highest address | BBH | 0 |
| Block type | BT | C |
| Close flag (positioning of file after close) | CF | N |
| Length in characters of record after trailer count field (T type records only) | CL | n/a |
| Conversion mode | CM | n/a |
| Beginning character position of trailer count field, numbered from zero (T type records only) | CP | n/a |
| Length field (D type records) or trailer count field (T type records) is binary | CI | n/a |
| Type of information to be listed in dayfile | DFC | 3 |
| Type of information to be listed in error file | EFC | 0 |
| Error options | EO | AD |
| Trivial error limit | ERL | 0 |
| Fatal Flush | FF | 0 |
| Length in characters of an F or Z type record (same as MRL) | FL | n/a |
| File organization | FO | WA |
| Character length of fixed header for T type records | HL | n/a |
| Length of user's label area (number of characters) | LBL | n/a |
| Logical file name | LFN | †† |
| Length in characters of record length field (D type records) | LL | n/a |
| Beginning character position of record length, numbered from zero (D type records) | LP | n/a |
| Label type | LT | n/a |
| Maximum block length in characters | MBL | n/a |
| Minimum block length in characters | MNB | n/a |

(Continued on next page)

Table E-3. NOS FORTRAN Default FIT Fields for Direct Access Formatted and Unformatted I/O

(Continued from previous page)

| FIT Field Meaning | Mnemonic | Value |
|---|---|---|
| Minimum record length in characters | MNR | n/a |
| Maximum record length in characters | MRL | n/a |
| Multiple of characters per K, E type block | MUL | n/a |
| Open flag (positioning of file after open) | OF | N |
| Padding character for K, E type blocks | PC | n/a |
| Processing direction | PD | IO |
| Number of records per K type block | RB | n/a |
| Record mark character (R records) | RMK | n/a |
| Record type | RT | U |
| Length field (D type records) or trailer count field (T type records) has sign overpunch | SB | NO |
| Suppress buffering | SBF | n/a |
| Suppress read ahead | SPR | n/a |
| Character length of trailer portion of T type records | TL | n/a |
| User label processing | ULP | NO |
| End of volume flag (positioning of file at volume CLOSEM time) | VF | U |

†For use with FMA, buffer is determined by CYBER Record Manager using the record length.

††For FMA, set by a FILE command and by the EXECUTE_MIGRATION_TASK command, PROGRAM or OPEN statements. For regular use on CYBER 170, set by PROGRAM or OPEN statements, or by an execution control statement or a FILE command.

# NOS and NOS/VE Similarities/Differences Summary F

This discussion provides a summary of similarities and differences between processing on the old CDC CYBER series computer systems and the new CYBER computer systems that support dual-state processing. The old systems support either NOS or NOS/BE only. The new systems support NOS/VE (called the virtual state) and NOS (called the real state). The similarities and differences of the systems are presented in tables as follows:

| | |
|---|---|
| Table F-1 | CPU |
| Table F-2 | Memory |
| Table F-3 | Peripheral Processor |
| Table F-4 | Operating System |

TABLE F-1. CPU Similarities/Differences Summary

| All Other CYBER 170 Models | Dual-State CYBER 170 Models |
|---|---|
| 60-bit word | 64-bit word |
| Word addressing (10 bytes per word; 6 bits per byte) | Byte/word addressing (8 bytes per word; 8 bits per byte) |
| 8 X registers (60 bits) | 16 X registers (64 bits) |
| 8 B registers (18 bits) | No B registers |
| 8 A registers | 16 A registers (48 bits) for store/ load instructions. |
| 1's complement arithmetic | 2's complement arithmetic with NOS/VE; 1's complement arithmetic with NOS |
| CYBER 170 real state instruction set | CYBER 170 real state and virtual state instructions are enabled using instruction set mode bits Virtual Machine Identifier (VMID) in virtual state exchange package. |
| Register-to-register operations | Register-to-register operations |

Table F-2. Memory Similarities/Differences Summary

| All Other CYBER 170 Models | Dual-State CYBER 170 Models |
|---|---|
| Maximum 131K word user address space | 4096 times 2**31-byte user virtual address space |
| RA/FL relocation | Hardware-segmented memory (maximum 4096 segments per user address space) |
| 17-bit word address within RA/FL defined address space | Two-part virtual address<br>- Segment number (12 bits)<br>- Signed byte offset into segment (32 bits) space |
| 262K word maximum system executable memory | 64 M byte (potentially 2**31 byte) executable real memory |
| Memory moves and swapping to manage memory | Hardware paging and swapping to manage memory |

Table F-3. Peripheral Processor Similarities/Differences Summary

| All Other CYBER 170 Models | Dual-State CYBER 170 Models |
|---|---|
| Up to 2x10 12-bit peripheral processor units (PPU's) | Up to 4x5 16-bit peripheral processors (PP's) |
| Up to 2x12 12-bit channels | Up to 6x4 12/16-bit channels |
| Executes 12-bit PPU code | Executes 12-bit, 16-bit, or mixture, PP code (upward compatible with other CYBER 170 models) |
| Memory size is 4K x 12 bits | Memory size is 4K x 16 bits |
| 12-bit wide data channels | 12-bit and/or 16-bit wide data channels |
| 60-bit access to central memory | 64-(4x16 bits) and 60-(5x12 bits) bit access to central memory |
| 18-bit central memory address for PPU read/write operations | 28-bit central memory address for PP read/write operations |
| Real central memory addressing | Real central memory addressing |
| 500-ns major cycle time | 250-ns major cycle time |
| 16-word deadstart panel | 16-word deadstart panel and 512-word read-only memory usable at deadstart |

Table F-4. Operating System Similarities/Differences Summary

| All Other CYBER 170 Models | Dual-State CYBER 170 Models |
|---|---|
| No shared memory among user address spaces (defined by its RA/FL) | Segments can be shared among user address spaces (code and data sharing possible) |
| Code and data mixed within user's address space | Segments can be accessed for read, write, execute, or a combination of operations; code can be shared on a global basis |
| Exchange operation to go to CPU Monitor | Exchange operation to go to CP Monitor |
| CPU supports CYBER 170 real state instruction set | CP supports coexisting CYBER 170 real state and virtual state instruction sets. VMID field, within the virtual state exchange package, is used to switch between real and virtual state instruction sets. The environment for NOS is established within the virtual state job space and then state switching can be accomplished by an exchange or trap operation and CALL or RETURN instructions. Real state external interrupts are supported and handled within the real state environment. |
| System runs at system control points in PPU's; CPU Monitor routes RA+1 requests | Most NOS/VE system code runs within user address space and obeys the same loading and linking conventions.<br>- Levels of system code are protected by a hardware-supported hierarchical ring mechanism from less capable code modules (15 ring levels are provided).<br>- System code can be directly called by a CALL instruction similar to a return jump without software assistance. |
| Subsystems (that is, IAF, MAG, and so forth) are protected by RA/FL mechanisms from each other or the user and can be called only via CPU Monitor. | Subsystems are protected by hardware-supported (key/lock) mechanisms from each other and are directly callable by user code without software assistance. |

# Index

## C

## D

## E

## F

## G

## H



## I



## J

## K



## L



## M

## N



## O

## P

## Q



## R

## S

## T



## U

## V



## W



## X



## Y



## Z

# Migration From NOS to NOS/VE, Tutorial/Usage 60489503 F

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?**

__ SCL Advanced File Management Usage
__ FORTRAN Language Definition Usage
__ COBOL Usage

What programming languages do you use? _____

Which are helpful to you? __ Quick Index (inside cover) __ Character Set appendix
__ Related Manuals appendix
__ Other: _____

**How Do You Like This Manual?** Check those that apply.

| Yes | Somewhat | No | |
|---|---|---|---|
| __ | __ | __ | Is the manual easy to read (print size, page layout, and so on)? |
| __ | __ | __ | Is it easy to understand? |
| __ | __ | __ | Is the order of topics logical? |
| __ | __ | __ | Are there enough examples? |
| __ | __ | __ | Are the examples helpful? (__ Too simple __ Too complex) |
| __ | __ | __ | Is the technical information accurate? |
| __ | __ | __ | Can you easily find what you want? |
| __ | __ | __ | Do the illustrations help you? |
| __ | __ | __ | Does the manual tell you what you need to know about the topic? |

**Comments?** If applicable, note page number and paragraph.

Continue on other side

**Would you like a reply?** __ Yes __ No

**From:**

Name _____ Company _____

Address _____ Date _____

_____ Phone No. _____

_____

Please send program listing and output if applicable to your comment.

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MN.

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

**Technology and Publications Division**
Mail Stop: SVL104
P.O. Box 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD

FOLD

# Quick Index

## Corresponding NOS-NOS/VE Commands

| NOS Command | NOS/VE Command and Comments | | Page |
|---|---|---|---|
| APL | APL | | 7-17 |
| ASSIGN | REQT | (REQUEST_TERMINAL lfn) | † |
| ATTACH | ATTF | (ATTACH_FILE) | 4-13 |
| | | Implicit Attach | 4-14 |
| CATLIST | DISC | (DISPLAY_CATALOG) | 4-15 |
| CATLIST,FN= | DISCE | (DISPLAY_CATALOG_ENTRY) | 4-22 |
| CHANGE | CREFP | (CREATE_FILE_PERMIT) | 4-16 |
| | DELFP | (DELETE_FILE_PERMIT) | 4-19 |
| | CHACE | (CHANGE_CATALOG_ENTRY) | 4-20 |
| COBOL5 | COBOL | Calls COBOL Compiler | 7-3 |
| COPY or COPYEI | COPF | (COPY_FILE) | 4-15 |
| DAYFILE | DISL | (DISPLAY_LOG) | 4-28 |
| | CREFC | (CREATE_FILE_CONNECTION) | 4-28 |
| Example: CREFC $ECHO lfn | | | |
| DEFINE | CREF | (CREATE_FILE) | 4-11 |
| | | Implicit Create | 4-12 |
| DISPLAY | DISV | (DISPLAY_VALUE) | 4-33 |
| FILE | SETFA | (SET_FILE_ATTRIBUTES) | 4-22 |
| | CHAFA | (CHANGE_FILE_ATTRIBUTES | 4-22 |
| | ATTF | (ATTACH_FILE) | 4-13 |
| FORM | FMU | File Management Utility | 11-5 |
| FTN5 | FORTRAN or FTN | | 7-1 |
| GET | ATTF | Also COPY_FILE (COPF) | 4-13 |
| Example: COPF I=$USER.permfile O=lfn | | | |
| LGO | LGO | Binary file | 7-8 |
| | | EXET (EXECUTE_TASK) | 7-8 |
| LIBGEN | CREATE_OBJECT_LIBRARY (CREOL) | | 8-1 |
| MERGE | MERGE | | † |
| NOTE | PUTL | (PUT_LINE) | 4-34 |
| PERMIT | CREFP | (CREATE_FILE_PERMIT) | 4-16 |
| | DELFP | (DELETE_FILE_PERMIT) | 4-19 |
| | CRECP | (CREATE_CATALOG_PERMIT) | 4-16 |
| | DELCP | (DELETE_CATALOG_PERMIT) | 4-20 |
| PURGE | DELF | (DELETE_FILE) | 4-12 |
| REPLACE | COPF | (COPY_FILE) | |
| Example: COPF I=lfn O=$USER.permfile | | | 4-15 |
| RETURN | DETF | (DETACH_FILE) | 4-14 |
| SAVE | COPF | (COPY_FILE) | |
| Example: COPF I=lfn O=$USER.permfile | | | 4-15 |
| SORT5 | SORT | | † |
| SUBMIT | SUBJ | (SUBMIT_JOB) | 9-3 |

Notes: † Not described in this manual.
lfn Represents the temporary file name.
permfile Represents the file name of a permanent file in your master catalog.

# Other Useful NOS/VE Commands

| Miscellaneous Commands Grouped as Applicable | Page |
|---|---|
| COLLECT_TEXT (COLT) | 4-31 |
| CREATE_INTERSTATE_CONNECTION (CREIC) | 4-26 |
| EXECUTE_INTERSTATE_COMMAND (EXEIC) | 4-26 |
| DELETE_INTERSTATE_CONNECTION (DELIC) | 4-26 |
| CREATE_FILE_CONNECTION (CREFC) | 4-28 |
| DELETE_FILE_CONNECTION (DELFC) | 4-30 |
| DISPLAY_FILE_CONNECTION (DISFC) | 4-31 |
| SET_FILE_ATTRIBUTES (SETFA) | 4-22 |
| CHANGE_FILE_ATTRIBUTES (CHAFA) | 4-24 |
| DISPLAY_FILE_ATTRIBUTES (DISFA) | 4-25 |
| RECEIVE_FILE (RECF) | 11-3 |
| SEND_FILE (SENF) | 11-4 |
| OPEN_FILE_MIGRATION_AID (OPEFMA) | 12-11 |
| OPEN_170_STATE (OPEIS) | 12-12 |
| EXECUTE_COMMAND (EXEC) | 12-12 |
| EXECUTE_MIGRATION_TASK (EXEMT) | 12-13 |
| COLLECT_FILE_DESCRIPTION (COLFD) | 12-37 |
| MIGRATE_FILE (MIGF) | 12-38 |
| CREATE_170_REQUEST | 12-54 |
| CHANGE_170_REQUEST | 12-63 |
| DISPLAY_TAPE_LABEL_ATTRIBUTES | 12-70 |
| DETACH_FILE | 12-78 |


CONTROL DATA