

CYBIL for NOS/VE File Interface

CD
CONTROL
DATA



Usage

60464114

Command Index

AMP\$ABANDON_KEY_	
DEFINITIONS	10-67
AMP\$ACCESS_METHOD_	
PROCEDURE	D-13
AMP\$ADD_TO_FILE_	
DESCRIPTION	D-11
AMP\$_APPLY_KEY_	
DEFINITIONS	10-65
AMP\$CLOSE	7-5
AMP\$COPY_FILE	11-9
AMP\$CREATE_KEY_	
DEFINITION	10-58
AMP\$DELETE_KEY	10-35
AMP\$DELETE_KEY_	
DEFINITION	10-64
AMP\$FETCH	6-15
AMP\$FETCH_ACCESS_	
INFORMATION	7-16
AMP\$FETCH_FAP_	
POINTER	D-16
AMP\$FILE	6-5
AMP\$FLUSH	9-31
AMP\$GET_DIRECT	9-21
AMP\$GET_FILE_	
ATTRIBUTES	6-13
AMP\$GET_KEY	10-25
AMP\$GET_KEY_	
DEFINITIONS	10-75
AMP\$GET_NEXT	9-23
AMP\$GET_NEXT_KEY	10-28
AMP\$GET_NEXT_PRIMARY_	
KEY_LIST	10-81
AMP\$GET_PARTIAL	9-26
AMP\$GET_PRIMARY_KEY_	
COUNT	10-78
AMP\$GET_SEGMENT_	
POINTER	8-6
AMP\$OPEN	7-2
AMP\$PUT_DIRECT	9-33
AMP\$PUT_KEY	10-16
AMP\$PUT_NEXT	9-35
AMP\$PUT_PARTIAL	9-37
AMP\$PUTREP	10-31
AMP\$REPLACE_KEY	10-33
AMP\$RETURN	2-7
AMP\$REWIND	9-17
AMP\$SEEK_DIRECT	9-11
AMP\$SELECT_KEY	10-74
AMP\$SET_FILE_INSTANCE_	
ABNORMAL	D-20
AMP\$SET_SEGMENT_	
EOI	8-18
AMP\$SET_SEGMENT_	
POSITION	8-20
AMP\$SKIP	9-18
AMP\$SKIP_TAPE_MARKS	4-9
AMP\$START	10-21
AMP\$STORE	6-8
AMP\$STORE_FAP_	
POINTER	D-15
AMP\$VALIDATE_CALLER_	
PRIVILEGE	D-8
AMP\$WRITE_END_	
PARTITION	9-39
AMP\$WRITE_TAPE_	
MARK	4-12
CLP\$CREATE_FILE_	
CONNECTION	2-9
CLP\$DELETE_FILE_	
CONNECTION	2-10
IFP\$FETCH_TERMINAL	5-11
IFP\$GET_DFLT_TERM_	
ATTRIBUTES	5-4
IFP\$GET_TERMINAL_	
ATTRIBUTES	5-7
IFP\$STORE_TERMINAL	5-9
IFP\$TERMINAL	5-2
PFP\$ATTACH	3-34
PFP\$CHANGE	3-10
PFP\$DEFINE	3-4
PFP\$DEFINE_CATALOG	3-14
PFP\$DELETE_CATALOG_	
PERMIT	3-28
PFP\$DELETE_PERMIT	3-24
PFP\$PERMIT	3-20
PFP\$PERMIT_CATALOG	3-25
PFP\$PURGE	3-7
PFP\$PURGE_CATALOG	3-15
RMP\$GET_DEVICE_	
CLASS	2-3
RMP\$REQUEST_NULL_	
DEVICE	2-5
RMP\$REQUEST_TAPE	4-3
RMP\$REQUEST_	
TERMINAL	5-6

CYBIL for NOS/VE File Interface

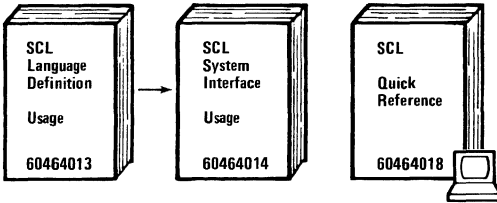
Usage

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.

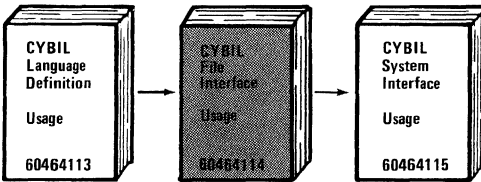
Publication Number 60464114

Related Manuals

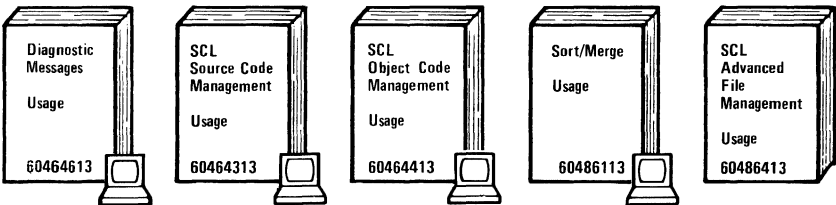
Background (Access as Needed):




CYBIL Manual Set:



Additional References:



→ indicates reading sequence.

 means available online.

Copyright 1983, 1984 by Control Data Corporation
All rights reserved.
Printed in the United States of America.

Manual History

Revision B reflects the release of NOS/VE 1.1.1 at PSR level 613. It was printed July 1984.

The changes consist of rewrites of chapters 1 and 3 to improve usability, new formats for all CYBIL procedures, additions to indexed sequential files, major editorial changes throughout, and inclusion of new CYBIL types.

Because changes to this manual are extensive, individual changes are not marked. This edition obsoletes all previous editions.

Previous Revision	System Level	Date
A	1.0.2	February 1984



Contents

How To Use This Manual ... 7

- Audience For This Manual ... 7
- Organization 8
- Conventions 9
- Additional Related
Manuals 10

Part I

Introduction

How to Use Interface

- Calls** 1-1
 - Using File Interface
 - Procedures 1-1
 - System Naming
 - Convention 1-9
 - Procedure Call Description
 - Format 1-10

Part II

Assigning Files To Devices

Local File Management 2-1

- Specifying the Device
- Class of a New File 2-1
- Null Device Class 2-4
- Returning a Local File 2-6
- File Connections 2-8

Mass Storage File

- Management** 3-1
 - File and Catalog Paths 3-1
 - File Cycles 3-3
 - File Subcatalogs 3-13
 - Access Modes 3-16
 - File Access Log 3-16
 - Access Control Entries 3-17
 - Attaching a Permanent
File 3-29

Tape Management 4-1

- Tape File Requests 4-1
- Tape File Attributes 4-6
- Tape File Positioning 4-7

Terminal Management 5-1

- Default Terminal
- Attributes 5-1
- Terminal File Requests 5-5
- Changing Terminal
- Attribute Values
- After the File
- Is Open 5-8
- Terminal Attributes 5-12
- Special Considerations
- for Terminal File
- Processing 5-26
- Terminal Output 5-30
- Terminal Conditions 5-33

Part III

Assigning File Data

Defining File Attributes ... 6-1

- Defining New File
- Attributes 6-1
- Defining Old File
- Attributes 6-3
- Defining Attributes
- for an Open File 6-4
- Attribute Definition
- Calls 6-4
- Retrieving File
- Attributes 6-9
- File Attribute
- Descriptions 6-16

File Opening and Closing 7-1

- File Identifiers 7-1
- Access Validation 7-6
- Error Exit Procedure 7-7
- File Sharing 7-8

Accessing a File as a Memory Segment 8-1

- CYBIL Data Storage 8-1
- Virtual Memory Access ... 8-3
- Segment Attributes 8-4
- Segment Pointer..... 8-5
- Sharing a Segment
 - Access File 8-15

Accessing Sequential and Byte Addressable Files 9-1

- Logical File Structure 9-1
- Working Storage Area 9-1
- Record Types 9-2
- File Blocking 9-4
- Sequential Record
 - Access 9-7
- Random Record
 - Access 9-9
- File Positioning 9-14
- Reading Records 9-20
- Writing Records 9-29

Accessing Indexed Sequential Files 10-1

- Primary Keys 10-1
- Indexed Sequential
 - File Structure 10-3
- Processing an Existing Indexed Sequential
 - File 10-18
- Monitoring the Index Levels in an Indexed Sequential
 - File 10-37
- Indexed Sequential File
 - Example 10-39
- Alternate Keys 10-47

File Copying 11-1

- Sequential File
 - Organization to Sequential File
 - Organization 11-2
- Sequential File
 - Organization to Indexed Sequential
 - File Organization 11-3
- Byte Addressable
 - File Organization to Byte Addressable
 - File Organization 11-4
- Indexed Sequential
 - File Organization to Indexed Sequential
 - File Organization 11-5
- Indexed Sequential File
 - Organization to Sequential File
 - Organization 11-7
- List File Copying 11-8
- File Copy Example 11-11

Part IV

Appendixes

Glossary A-1

ASCII Character Set B-1

Constant and Type Declarations C-1

File Access Procedures .. D-1

Collation Tables for Indexed Sequential Files E-1

Common Procedures F-1

Index Index-1

About This Manual

This manual describes CONTROL DATA® CYBIL procedure calls that interface between the CDC® Network Operating System/Virtual Environment (NOS/VE) and CYBIL programs. CYBIL is the implementation language of NOS/VE.

NOS/VE provides a program interface written in the CYBIL language through which CYBIL programs can interface to the operating system. This program interface is comprised of CYBIL procedures which are designed to be used in CYBIL programs. These CYBIL procedures are topically divided for presentation in two manuals: the CYBIL System Interface manual, and this, the CYBIL File Interface manual.

Audience

This manual is written as a reference for CYBIL programmers. It assumes that the reader knows the CYBIL programming language as described in the CYBIL Language Definition manual.

To use the procedure calls described in this manual, the programmer must copy decks from a system library. Although this manual provides a brief description of the commands required to copy procedure declaration decks, the SCL Source Code Management manual contains the complete description.

This manual also assumes that the reader has used the System Command Language (SCL). You can perform many system functions described in this manual using either SCL commands or CYBIL procedure calls. Commands referenced in this manual are SCL commands. For a description of SCL command syntax, see the SCL Language Definition manual; for individual SCL command descriptions, see the SCL System Interface manual.

Other manuals that relate to this manual are shown on the Related Manuals page.

Organization

The CYBIL File Interface manual is divided into four parts:

- Introduction
- Assigning Files to Devices
- Accessing File Data
- Appendixes

The first part is an introduction to the use of system-supplied file interface calls. You should read the introduction first.

Each of the chapters in the second and third parts describes a certain function. You can read these chapters in any order. For example, if you do not plan to use tape files, you can skip the chapter on tape management.

The Assigning Files to Devices part describes calls to assign files to device classes. Separate chapters describe mass storage, tape, and interactive terminal assignment.

The Accessing File Data part describes calls used to access files regardless of their device assignment. Separate chapters describe file attribute definition, opening and closing files, and reading and writing file data.

The appendixes provide supplementary information:

- Appendix A Glossary.
- Appendix B ASCII character set.
- Appendix C System-defined type and constant declarations used by file interface procedures.
- Appendix D Description of the use and creation of file access procedures (FAPs).
- Appendix E Description of collation table creation for indexed sequential files.
- Appendix F Common procedures.

This manual is part of the CYBIL manual set. Besides this manual, the CYBIL manual set includes the following manuals:

- The CYBIL Language Definition manual that defines the CYBIL language in detail.
- The CYBIL System Interface manual that describes the NOS/VE-supplied system interface CYBIL procedures.

Conventions

- boldface** Within formats, procedure names are shown in boldface type. Required parameters are also shown in boldface.
- italics* Within formats, optional parameters are shown in italics.
- blue Within interactive terminal examples, user input is shown in blue.
- UPPERCASE Within formats, uppercase letters represent reserved words; they must appear exactly as shown in the format.
- lowercase Within formats, lowercase letters represent names and values that you supply.
- examples Examples are printed in a typeface that simulates computer output. They are shown in lowercase, unless uppercase characters are required for accuracy.
- numbers All numbers are base 10 unless otherwise noted.

Additional Related Manuals

Each procedure call description lists the exception conditions that the procedure can return. The message template and condition code associated with each condition is listed in the Diagnostic Messages for NOS/VE manual (publication number 60464613).

Ordering Manuals

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

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

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
Publications and Graphics Division ARH219
4201 Lexington Avenue North
St. Paul, Minnesota 55112

Please indicate whether you would like a written response.

How to Use File Interface Calls **1**

- Using File Interface Procedures 1-1
 - Copying Procedure Declaration Decks 1-3
 - Expanding a Source Program 1-4
 - Calling a File Interface Procedure 1-6
 - Parameter List 1-6
 - Checking the Completion Status 1-7
 - Exception Condition Information 1-7
- System Naming Convention 1-9
- Procedure Call Description Format 1-10
 - Parameter Description Format 1-10



How to Use File Interface Calls 1

NOS/VE provides a set of CYBIL procedures by which programs can request system services. A system service is a function which supplies information and capabilities to application programs. The functions are supported by the operating system. This manual describes the file interface portion of the NOS/VE-supplied CYBIL procedures. It provides the CYBIL programmer with the information required to make calls to file interface procedures in CYBIL programs.

Using File Interface Procedures

Each CYBIL file interface procedure resides as an externally referenced (XREF) procedure declaration in a deck on a system source library. In general, to use a file interface procedure, you must include the following statements in your CYBIL source program:

- A Source Code Utility (SCU) *COPYC directive to copy the XREF procedure declaration from a system source library.
- Statements to declare, allocate, and initialize actual parameter variables as needed.
- The procedure call statement.
- An IF statement to check the procedure completion status which is returned in the procedure's status variable.

Figure 1-1 lists a source program that illustrates use of a file interface procedure. System-defined names are shown in uppercase letters; user-defined names in lowercase letters.

```

MODULE example1;

{ Directives to copy the XREF procedure declarations.}

*copyc rmp$get_device_class
*copyc rmp$request_null_device

{ This procedure returns the device class of the file }
{ and a status record to the caller.}

PROCEDURE get_device_class
  (lfn: amt$local_file_name;
   VAR class_returned: rmt$device_class;
   VAR status: ost$status);

{ Parameter declarations }

  VAR
    device_assigned: boolean;

{ Procedure call statement }

RMP$GET_DEVICE_CLASS (lfn, device_assigned,
  class_returned, status);

{ Status record check. }

IF NOT status.NORMAL THEN
  RETURN;
IFEND;

IF device_assigned = FALSE THEN
  RMP$REQUEST_NULL_DEVICE (lfn, status);
  IF NOT status.NORMAL THEN
    RETURN;
  IFEND;
  class_returned := rmc>null_device;
IFEND;

PROCEND get_device_class;
MODEND example1;

```

Figure 1-1. File Interface Call Example

The following paragraphs describe in greater detail the SCU directives and CYBIL statements required to use a file interface procedure.

Copying Procedure Declaration Decks

To use a file interface procedure in a CYBIL module, the module must include an SCU `*COPYC` directive to copy the externally referenced procedure from a system library. The XREF procedure declarations for all file interface calls except the indexed sequential file calls described in chapter 10 are stored as decks in the source library file `$$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE`. The indexed sequential file procedure declarations are stored as decks in the source library file `$$SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE`.

The deck containing the procedure declaration has the same name as the procedure. For example, the `RMP$GET_DEVICE_CLASS` procedure is declared in a deck named `RMP$GET_DEVICE_CLASS`.

As shown in figure 1-1, the `*COPYC` directives begin in column one, specify the name of the deck to be copied, and, in this example, follow the `MODULE` statement. You will need only one `*COPYC` directive for calls to the same file interface procedure in your CYBIL module regardless of how many times the procedure is called. For instance, if the module in figure 1-1 had called the `RMP$GET_DEVICE_CLASS` procedure more than one time, the one `*COPYC` directive to copy the XREF `RMP$GET_DEVICE_CLASS` procedure deck would suffice.

For more information about the `*COPYC` directive, see the SCL Source Code Management manual.

Procedure declaration decks list the parameters and their valid CYBIL types that must be listed on a call to a file interface procedure. When a CYBIL program is being compiled, the parameters on the call to the file interface procedure are verified with the parameters and parameter types listed in the procedure's XREF procedure declaration. If the parameters on the call to the file interface procedure do not match the parameters and the parameter's required type as defined in the procedure declaration, the program compilation will fail. After the module in figure 1-1 is compiled, the XREF procedure declaration will be included in the source listing.

An example of a procedure declaration deck is found later in this chapter under the subheading, Calling a File Interface Procedure.

In this manual, the required parameters as well as each parameter's required type is listed in the individual procedure call description format for each file interface procedure. The parameter types for all CYBIL file interface procedures are listed alphabetically in appendix C.

Expanding a Source Program

A CYBIL source program that calls one or more file interface procedures must be expanded through commands provided in the Source Code Utility (SCU). Expanding the program through SCU generates the source code to be compiled.

The SCU process of expanding requires that the source program exist as one or more decks on an SCU library. The contents of a file containing a CYBIL module are transferred onto a deck when you issue the `CREATE_DECK` subcommand within an SCU session. An example of how to do this is shown in figure 1-2.

To expand a CYBIL source program that calls file interface procedures, you use the SCU `EXPAND_DECK` subcommand. You list the name of the decks to be expanded on the `DECK` parameter, and you list the `SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE` file, which contains the XREF procedure decks for all file interface procedures, as the `ALTERNATE_BASE` parameter on the same `EXPAND_DECK` subcommand. SCU then processes the specified decks, copying any XREF decks named on `*COPYC` directives into the CYBIL source program.

If the CYBIL program uses indexed sequential file calls, the file `SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE` must also be specified on the `ALTERNATE_BASE` parameter. Figure 1-2 shows a command sequence.

```

1. /create_source_library
2. /scu base=result result=$user.my_library
3. sc/create_deck deck=my_program modification=mod0 ..
   sc../source=source_file
4. sc/expand_deck deck=my_program ..
   sc../alternate_base= ..
   sc../($system.cybil.osf$program_interface,..
   sc../$system.common.psf$external_interface_source)
5. sc/quit write_library=true
6. /cybil i=compile l=listing lo=(x, r, a)

```

Figure 1-2. Source Text Preparation Example

The command sequence in figure 1-2 performs the following tasks:

1. Creates an empty source library on the default file RESULT.
2. Calls SCU. The base library is the empty library on file RESULT that was created in step 1; the result library will be written on the user's permanent file, MY_LIBRARY, at the end of the SCU session.
3. Creates a deck named MY_PROGRAM. The deck was created from the local file, SOURCE_FILE which contained the CYBIL program.
4. Expands the MY_PROGRAM deck. Expanding the MY_PROGRAM deck will process any *COPYC directives included in the source program. Any decks specified by the *COPYC directives will be copied from the library files OSF\$PROGRAM_INTERFACE or PSF\$EXTERNAL_INTERFACE_SOURCE which are listed on the ALTERNATE_BASE parameter. The expanded text is written on the default file COMPILE.
5. Ends SCU processing. The WRITE_LIBRARY=TRUE parameter indicates that the library is to be written on the result library file. (This is not required; the expanded source text remains available on the COMPILE file even if no result library is written.)
6. Calls the CYBIL compiler to compile the text on file COMPILE and write a source listing on file LISTING.

For more information on creating source libraries and decks and on expanding decks, see the SCL Source Code Management manual.

Calling a File Interface Procedure

A call to a file interface procedure has the same format as any CYBIL procedure call. In general, a CYBIL procedure call statement has the following format:

```
procedure_name (parameter_list);
```

For more information on CYBIL procedure calls, see the CYBIL Language Definition manual.

Parameter List

A procedure parameter list provides the procedure with input values and the locations where it is to store output values. You can specify an input value as the value itself or as a variable containing the value.

NOTE

All parameters on a procedure call are required. You must specify a value or variable for each parameter in the parameter list.

CYBIL performs type checking on the variables and values specified in a procedure parameter list. It compares the parameters on the procedure call with the parameter types listed in the XREF procedure declaration. Therefore, to make a successful call to a file interface procedure, the parameters on the procedure call must conform to the parameter types specified in the procedure declaration deck.

For example, the procedure declaration for the RMP\$GET_DEVICE_CLASS procedure is as follows:

```
PROCEDURE [XREF] rmp$get_device_class
  (local_file_name: amt$local_file_name;
   VAR device_assigned: boolean;
   VAR device_class: rmt$device_class;
   VAR status: ost$status);
```

This declaration indicates that a call to the procedure must specify four parameters in its parameter list. The first parameter must specify an input value of type AMT\$LOCAL_FILE_NAME; the second parameter must specify a variable of type BOOLEAN; the third parameter must specify a variable of type RMT\$DEVICE_CLASS; and the fourth parameter must specify a variable of type OST\$STATUS. The “VAR” listed with the last three parameters indicates that these parameters are treated as output parameters by the RMP\$GET_DEVICE_CLASS procedure; that is, values will be returned to these parameters by the procedure.

All parameter types as well as the valid parameter values are listed in the individual parameter descriptions for each file interface procedure described in this manual.

For more information on declaring and assigning values to variables, see the CYBIL Language Definition manual.

Checking the Completion Status

The last parameter on a file interface procedure call must be a status variable (type OST\$STATUS). Unlike the status parameter on SCL commands, the status parameter on file interface calls is required, not optional. When the procedure completes, NOS/VE returns the completion status of the procedure in the specified status variable.

The program should check the completion status returned immediately after the procedure call. If the NORMAL field of the status variable is TRUE, the procedure completed normally. If the NORMAL field is not TRUE (that is, FALSE), the procedure completed abnormally.

For example, the following program fragment uses a status variable named STATUS. Immediately after the RMP\$GET_DEVICE_CLASS call, an IF statement checks the value of the boolean field of the status record (STATUS.NORMAL). If its value is false (NOT STATUS.NORMAL), the procedure terminates.

```

rmp$get_device_class (local_file_name,device_assigned,
    device_class, status);
IF NOT status.NORMAL THEN
    RETURN;
IFEND;

```

Exception Condition Information

When the procedure completes abnormally, NOS/VE returns additional information about the exception condition that occurred. The following variant fields of the record return condition information when the key field, NORMAL, is false:

identifier

Two-character string identifying the process that detected the error. Table 1-1 lists the identifiers returned by calls described in this manual.

condition

Exception condition code that uniquely identifies the condition (OST\$STATUS_CONDITION, integer). Each code can be referenced by its constant identifier as listed in the Diagnostic Messages manual.

text

String record (type OST\$STRING). The record has the following two fields:

size

Actual string length in characters (0 through 256).

value

Text string (256 characters).

NOTE

The text field does not contain the error message. It contains items of information that are inserted in the error message template if the message is formatted using this status variable.

If the NORMAL field of the status record is FALSE, the program determines its subsequent processing. For example, it could check for a specific condition in the CONDITION field or determine the severity level of the condition with a OSP\$GET_STATUS_SEVERITY procedure call. The CYBIL System Interface manual contains the description of OSP\$GET_STATUS_SEVERITY and other condition processing calls.

Table 1-1. Process Identifiers for File Interface Calls

Process Identifier	Process Function
AA	Advanced access method.
AM	Access method.
CL	Command language.
IF	Interactive file and terminal management.
OS	Operating system.
PF	Permanent file management.
RM	Resource management.

System Naming Convention

All identifiers defined by the NOS/VE file interface use the system naming convention. The system naming convention requires that all system-defined CYBIL identifiers have the following format:

idx\$name

Field	Description
-------	-------------

id	Two characters identifying the process that uses the identifier. Table 1-1 lists the identifiers used in this manual.
----	---

x	Character indicating the CYBIL element type identified.
---	---

x	Description
---	-------------

c	Constant.
---	-----------

e	Error condition.
---	------------------

p	Procedure.
---	------------

t	Type.
---	-------

\$	The \$ character indicates that Control Data defined the identifier.
----	--

NOTE

To ensure that each identifier you define differs from all Control Data-defined identifiers, avoid using the \$ character in your identifier. Each Control Data-defined identifier contains a \$ character.

name	A string of characters describing the purpose of the element the identifier represents.
------	---

For example, the identifier RMP\$GET_DEVICE_CLASS follows the system naming convention. Its process id is RM, for resource management. The P following the process id indicates that it is a procedure name. The string GET_DEVICE_CLASS describes the purpose of the procedure.

Procedure Call Description Format

Each of the remaining chapters of this manual describes a group of file interface procedures. Within the chapter are individual procedure call descriptions. Each procedure description uses the same format.

Each procedure description has the following subheadings:

Purpose	Brief statement of the procedure function.
Format	Procedure call format showing the parameter positional order followed by individual parameter descriptions.
Parameters	Descriptions of the parameters in the preceding format including the parameter's valid CYBIL type.
Condition Identifiers	List of condition identifiers returned by the procedure. The list is not all-inclusive; however, it lists conditions that are likely to be of interest to the procedure user.
Remarks	If present, additional information about procedure processing.

Parameter Description Format

Within each procedure call format description, each parameter description states the parameter function, the valid values for the parameter, and the parameter's valid CYBIL type. Appendix C contains an alphabetical listing of all parameter types for the CYBIL procedures described in this manual.

If the parameter type is a set of system-defined identifiers, the parameter description lists all possible identifiers in the set and their meanings.

If the variable type is a record, the parameter description describes each field in the record. It states the field name, its function, and its type.

- Specifying the Device Class of a New File 2-1
 - Overriding the Device Class 2-2
 - RMP\$GET_DEVICE_CLASS 2-3
- Null Device Class 2-4
 - RMP\$REQUEST_NULL_DEVICE 2-5
- Returning a Local File 2-6
 - AMP\$RETURN 2-7
- File Connections 2-8
 - System File Connections 2-8
 - CLP\$CREATE_FILE_CONNECTION 2-9
 - CLP\$DELETE_FILE_CONNECTION 2-10



Each file has the following characteristics:

- A local file name unique within the job.
- Assignment to a device class.
- A set of file attributes.

This chapter describes the assignment of a new file to a device class. Chapter 6, *Defining File Attributes*, describes the definition of file attributes.

The local file name identifies the file within the job. You can define a local file name when you specify a file reference on a command or call. Unless explicitly specified otherwise, the file is assigned to the default device class (mass storage) and the default file attribute set.

Specifying the Device Class of a New File

For purposes of clarity in this manual, a file is termed a new file if it has never been opened. When a task opens a new file, the system assigns the file to a device within its device class as follows:

- **Magnetic tape:** Assigns the file to tape devices.
- **Terminal:** Assigns the file to the interactive terminal.
- **Mass storage:** Assigns the file to a disk unit.

Before a task creates a new file, the new file can explicitly be associated with a device class by using one of the following commands or CYBIL calls:

- **Commands:** `CREATE_FILE`, `REQUEST_MAGNETIC_TAPE`, and `REQUEST_TERMINAL`.
- **Calls:** `PFPS$DEFINE`, `RMP$REQUEST_NULL_DEVICE`, `RMP$REQUEST_TAPE`, and `RMP$REQUEST_TERMINAL`.

If no command or call has associated the local file name with a device class when the file is opened, the system assigns the file to mass storage.

Overriding the Device Class

Although device assignment calls within a program can specify the default device class of a file, an SCL command issued for the file before the program is executed will always override the device class specified by calls within the program.

When NOS/VE opens a new file, it determines the file's device class as follows:

- If the file was created on a CREATE_FILE command, it has already been assigned to a mass storage device.
- If a REQUEST_MAGNETIC_TAPE or REQUEST_TERMINAL command has been issued for the file, NOS/VE assigns the file to the tape or terminal device class, respectively.
- If the program has issued one or more RMP\$REQUEST_TAPE, RMP\$REQUEST_TERMINAL, or RMP\$REQUEST_NULL_DEVICE calls for the file, the last call issued before the file is opened is effective.
- If no command or call has assigned the file to a device class, NOS/VE assigns the file to the mass storage device class when it opens the file.

For example, suppose a program contains an RMP\$REQUEST_TAPE call that specifies the local file name TAPE1. Suppose the following command is executed before the program:

```
REQUEST_TERMINAL FILE=TAPE1
```

The device class specified by the command, REQUEST_TERMINAL, overrides the device class specified by the call, RMP\$REQUEST_TAPE, in the CYBIL module. Therefore, when the task opens the file, the file is assigned to the interactive terminal device.

Once the file has been opened, the device class cannot be changed; however, the file may be deleted and the file name may then be associated with a different device class.

To determine the device class associated with a local file name, a task can call the RMP\$GET_DEVICE_CLASS procedure.

RMP\$GET_DEVICE_CLASS

Purpose Returns the device class for a file.

Format RMP\$GET_DEVICE_CLASS (**local_file_name**,
device_assigned, **device_class**, **status**)

Parameters **local_file_name**: amt\$local_file_name;
Local file name.

device_assigned: VAR of boolean;

Indicates whether the file has been assigned to a device.

TRUE

The file has been opened, or a CREATE_FILE, REQUEST_MAGNETIC_TAPE, or REQUEST_TERMINAL command has been issued for the local file name.

FALSE

The file has not yet been opened, and no CREATE_FILE, REQUEST_TERMINAL, or REQUEST_MAGNETIC_TAPE command has been issued for the local file name. In this case, the device_class value returned is always RMC\$MASS_STORAGE_DEVICE.

device_class: VAR of rmt\$device_class;

Device class.

RMC\$MASS_STORAGE_DEVICE

Mass storage.

RMC\$MAGNETIC_TAPE_DEVICE

Magnetic tape.

RMC\$TERMINAL_DEVICE

Interactive terminal.

RMC\$NULL_DEVICE

Null device.

status: VAR of ost\$status;

Status variable.

Condition Identifier None.

Null Device Class

Assignment of a file to the null device class means that data is discarded as it is written. Attempts to read data from the file always return an end-of-information status. A file is assigned to the null device class by calling the RMP\$REQUEST_NULL_DEVICE procedure.

When a task opens a null file for record access, it can issue get and put calls to the file. A get call returns normal status, but no data; a put call discards the data to be written and returns normal status. The file position returned depends on the call, as follows:

- A get call always returns AMC\$EOI.
- A full record put call returns AMC\$EOR.
- A partial record put call to write the beginning or middle part of a record returns AMC\$MID_RECORD.
- A partial record put call to write the end of a record returns AMC\$EOR.

When a task opens a null file for segment access, an AMP\$GET_SEGMENT_POINTER call returns a NIL pointer because the system does not assign a segment to the file.

An indexed sequential file cannot be assigned to the null device class.

A null file can be used for debugging purposes when a file reference is required in the code but any data access to the file is not appropriate. To discard any unwanted output generated by a call to a command or a CYBIL procedure, the file name \$NULL can be passed as the file reference.

RMP\$REQUEST_NULL_DEVICE

Purpose	Assigns a file to the null device class.
Format	RMP\$REQUEST_NULL_DEVICE (local_file_name, status)
Parameters	local_file_name: amt\$local_file_name; Local file name. status: VAR of ost\$status; Status variable.
Condition Identifier	None.
Remarks	<ul style="list-style-type: none">• The system ignores the request if the file is already assigned to a device.• A null file is a temporary file.• If the file is never opened, its association with the null device class has no effect.

Returning a Local File

A file remains assigned to a job until one of the following occurs:

- The file is closed while its `return_option` attribute value is `AMC$RETURN_AT_CLOSE`.
- The file is explicitly returned by a `DETACH_FILE` command or `AMP$RETURN` call.
- The job terminates.

Returning a temporary mass storage file, tape file, or terminal file ends the device assignment and discards the local file name and its file attribute set.

If the device class of the returned file is magnetic tape, its tape unit assignment ends; the tape volumes accessed via the local file name are no longer associated with that name.

When a temporary mass storage file is returned, all space allocated to the file is released and the file no longer exists. When a permanent mass storage file is returned, its space is not released and the file continues to exist; only its associated local file name is discarded. To access the permanent mass storage file again using the local file name, you attach the file using an `ATTACH_FILE` command or `PMP$ATTACH` call and specify the local file name. For more information on attaching a permanent mass storage file, see chapter 3, Mass Storage File Management.

AMP\$RETURN

Purpose	Detaches a file from a job. After the file is detached, it is no longer local to the job.
Format	AMP\$RETURN (local_file_name, status)
Parameters	<p>local_file_name: amt\$local_file_name; Name of a file local to the job.</p> <p>status: VAR of ost\$status; Status variable. The process identifier returned is AMC\$ACCESS_METHOD_ID.</p>
Condition Identifiers	<p>ame\$file_not_closed</p> <p>ame\$file_not_known</p> <p>ame\$ring_validation_error</p>
Remarks	<ul style="list-style-type: none"> • To return a file, all instances of open for the specified local file name must be closed. Standard files that reside in the \$LOCAL catalog (such as \$LIST) cannot be returned because those files always have an outstanding instance of open within a job. • If the file is assigned to mass storage, mass storage space associated with the file is released if the file is a temporary file; permanent file space is not affected. • If the file is assigned to an interactive terminal or tape unit, the assignment ends when the file is returned. However, returning a tape file does not decrement the tape unit reservation nor does it affect the information on the tape. If the tape unit was implicitly reserved by an AMP\$OPEN call, the reservation is implicitly released by a call to the AMP\$RETURN procedure.

File Connections

A file connection connects a subject file and a target file. The connection passes all data access calls for the subject file to the target file.

The CLP\$CREATE_FILE_CONNECTION call connects two files; the CLP\$DELETE_FILE_CONNECTION call removes a connection between files.

The system places no constraint on the file organization of either the subject or the target of the connection except that an indexed sequential file cannot be the subject of the connection. However, when creating a file connection, it is recommended that the file organization of both the subject and the target files be the same.

System File Connections

You cannot connect the system files with the following identifiers.

CLC\$CURRENT_COMMAND_OUTPUT

CLC\$JOB_COMMAND_INPUT

CLC\$JOB_INPUT

CLC\$JOB_OUTPUT

CLC\$NULL_FILE

The system initially connects its CLC\$JOB_COMMAND_RESPONSE file to either the CLC\$JOB_OUTPUT file (for an interactive job) or to the CLC\$NULL_FILE file (for a batch job). You cannot disconnect this initial connection.

CLP\$CREATE_FILE_CONNECTION

Purpose Connects a subject file to a target file.

Format CLP\$CREATE_FILE_CONNECTION (subject_file, target_file, status)

Parameters **subject_file**: amt\$local_file_name;
Subject file name.

target_file: amt\$local_file_name;
Target file name.

status: VAR of ost\$status;
Status variable.

Condition Identifiers cle\$circular_file_connection
cle\$improper_subject_file_name
cle\$improper_target_file_name
cle\$subject_cannot_be_connected

Remarks

- If a subject file is connected to more than one target file, calls are passed as follows:
 - A call to get data from the subject file is passed to the target file most recently connected.
 - A call to put data in the subject file is passed once for each connection.
 - An AMP\$GET_FILE_ATTRIBUTES, AMP\$FETCH, or AMP\$FETCH_ACCESS_INFORMATION call specifying the subject file returns the attribute values belonging to the first target file connected.
- A file connection takes effect immediately for all instances of open of the file.

CLP\$DELETE_FILE_CONNECTION

Purpose	Disconnects the subject file from the target file.
Format	CLP\$DELETE_FILE_CONNECTION (subject_file, target_file, status)
Parameters	subject_file: amt\$local_file_name; Subject file name. target_file: amt\$local_file_name; Target file name. status: VAR of ost\$status; Status variable.
Condition	cle\$connection_cannot_be_broken
Identifiers	cle\$improper_subject_file_name cle\$improper_target_file_name cle\$unknown_file_connection
Remarks	The disconnection is effective immediately.

Mass Storage File Management 3

File and Catalog Paths	3-1
Path Specification	3-2
File Cycles	3-3
Defining a File Cycle	3-3
PFP\$DEFINE	3-4
PFP\$PURGE	3-7
Changing File Entry Information	3-9
PFP\$CHANGE	3-10
File Subcatalogs	3-13
PFP\$DEFINE_CATALOG	3-14
PFP\$PURGE_CATALOG	3-15
Access Modes	3-16
File Access Log	3-16
Access Control Entries	3-17
Permit Selections	3-17
Share Requirements	3-18
Multiple Access Control Entries	3-19
PFP\$PERMIT	3-20
PFP\$DELETE_PERMIT	3-24
PFP\$PERMIT_CATALOG	3-25
PFP\$DELETE_CATALOG_PERMIT	3-28
Attaching a Permanent File	3-29
Attaching a File with PFP\$ATTACH or ATTACH_FILE	3-29
Attaching a File with a File Reference	3-30
Evaluating Attach Requests	3-30
File Cycle Busy Status	3-31
Wait Option	3-33
PFP\$ATTACH	3-34



Mass Storage File Management 3

The NOS/VE mass storage file system uses catalogs to organize and control access to mass storage files. A catalog is a data structure which contains files and subcatalogs.

Each mass storage file, temporary or permanent, is an entry in a catalog. All temporary mass storage files are entries in the \$LOCAL file catalog. Permanent mass storage files are entries in permanent file catalogs.

As a user of NOS/VE, you have a master catalog which is named for your user name. It contains any permanent files or subcatalogs that you create. You can define additional files and subcatalogs within each subcatalog. You are the owner of all files and subcatalogs defined in your master catalog.

Each job is provided with an empty \$LOCAL catalog. Files created in the \$LOCAL catalog are temporary; that is, they will be deleted when the job terminates. The \$LOCAL catalog cannot have subcatalogs and files in the \$LOCAL catalog that have only one file cycle.

File and Catalog Paths

To list a permanent mass storage file or subcatalog as a file entry parameter on a call to a file interface procedure, you must specify the path to the file or catalog. A catalog path contains the following elements.

- Family of users.
- List of one or more catalogs beginning with the master catalog.

A file path contains the following elements.

- Family of users.
- List of one or more catalogs beginning with the master catalog.
- Permanent file name.

The catalog sequence always begins with the master catalog. If the file or catalog is defined in the master catalog, the catalog sequence consists solely of the master catalog. If the file or catalog is defined in a subcatalog, the catalog sequence must include the appropriate subcatalog names.

For example, suppose user USERX in family FAMILY1 defines a subcatalog, SUB1. The path for subcatalog SUB1 is as follows:

FAMILY1 → USERX → SUB1

Next, USERX defines subcatalog SUBA in subcatalog SUB1. The path for subcatalog SUBA is as follows:

FAMILY1 → USERX → SUB1 → SUBA

Finally, USERX defines file FILE1 in subcatalog SUBA. The path for file FILE1 is as follows:

FAMILY1 → USERX → SUB1 → SUBA → FILE1

Path Specification

A call to a file interface procedure specifies a permanent file path within a variable of type PFT\$PATH.

The PFT\$PATH variable is a list of names, one per element of an adaptable array. The names specify the file path, including the family name, the master catalog name, the subcatalog names (if applicable), and, finally, the permanent file name.

The first name in the array must be the family name of the user. If the first name in the array is OSC\$NULL_NAME, the family name of the job is used.

The second name in the array must be the master catalog name. By convention, the master catalog name is the same as the name of the user. If the second name in the array is OSC\$NULL_NAME, the user name of the job is used.

Subsequent names in the array list the subcatalogs in the catalog path, if applicable. The last name in the array must be the name of the permanent file or subcatalog on which the operation is performed. The OSC\$NULL_NAME identifier cannot identify a subcatalog or permanent file.

The following constant identifiers are provided to allow symbolic reference to the initial elements of a path array:

- PFC\$FAMILY_NAME_INDEX: index to the family name.
- PFC\$MASTER_CATALOG_NAME_INDEX: index to the master catalog.
- PFC\$SUBCATALOG_NAME_INDEX: index to the first subcatalog in the path.

For example, if the name of the path array is PATH, the master catalog element of the array can be referenced as follows:

PATH[PFC\$MASTER_CATALOG_NAME_INDEX]

File Cycles

A mass storage file is defined by its file entry in a catalog. More than one version of the file can exist through the use of file cycles. Each file cycle is a separate version of the file and is uniquely defined by a cycle descriptor.

A file entry contains the following information:

- Permanent file name.
- File password.
- Access log selection.
- Account and project names for the file.

The information in the file entry applies to all cycles of the file.

A cycle descriptor contains the following information:

- Cycle number.
- Creation date and time for the cycle.
- Last modification date and time for the cycle.
- Last access date and time for the cycle.
- Cycle expiration date (determined by the retention period specified when the cycle is defined).

The information in a cycle descriptor applies only to that cycle.

Defining a File Cycle

To define a new permanent file cycle, you call the PFP\$DEFINE procedure. If the file entry parameter specified on the call does not exist, PFP\$DEFINE defines a file entry in the last subcatalog of the specified path and defines the initial cycle descriptor. If a file entry for the file already exists, PFP\$DEFINE only creates a new cycle descriptor.

A file cycle created by a PFP\$DEFINE call is assigned to the mass storage device class. When you make a call to PFP\$DEFINE, you specify a local file name by which the file cycle can be referenced within a job. The local file name is discarded when the file is returned (detached) or the job terminates.

Once the file cycle is defined, future attempts to attach it must specify the same file path. The file cycle definition is valid until the file cycle is purged.

A call to PFP\$PURGE removes one cycle of a file. To purge a file entry, you must call PFP\$PURGE for each cycle of the file.

PF\$DEFINE

Purpose Defines a permanent file cycle.

NOTE

To define a new file, you must have cycle permission for the catalog. To define a new cycle of an existing file, you must have cycle permission for the file.

Format PF\$DEFINE (lfn, path, cycle_selector, password, retention, log, status)

Parameters lfn: amt\$local_file_name;

Local file name.

path: pft\$path;

File path. The last name in the path list is the permanent file name.

cycle_selector: pft\$cycle_selector;

Permanent file cycle created.

Field	Content
-------	---------

cycle_option	Key field indicating the file cycle number.
--------------	---

PFC\$LOWEST_CYCLE

Creates a cycle numbered one less than the current lowest cycle number. If no cycles exist for the file, PF\$DEFINE creates cycle 1.

PFC\$HIGHEST_CYCLE

Creates a cycle numbered one greater than the current highest cycle number. If no cycles exist for the file, PF\$DEFINE creates cycle 1.

PFC\$SPECIFIC_CYCLE

Creates the cycle specified by the cycle_number field.

cycle_number	Cycle number (integer from 1 through PFC\$MAXIMUM_CYCLE_NUMBER, 999). If the cycle already exists, the procedure returns an error status (PFE\$DUPLICATE_CYCLE) without defining a new cycle.
--------------	---

password: pft\$password;

File password (1- through 31-character name). A blank password is the same as no password.

If the PFP\$DEFINE call creates a new file entry, it stores the specified password in the file entry. If PFP\$DEFINE creates a new cycle for an existing file entry, it compares the specified password with the password stored in the file entry. If the passwords do not match, the call returns abnormal status PFE\$INCORRECT_PASSWORD.

retention: pft\$retention;

Cycle retention period in days (1 through PFC\$MAXIMUM_RETENTION, 999; PFC\$MAXIMUM_RETENTION indicates infinite retention).

log: pft\$log;

Log option. If the PFP\$DEFINE call creates a new cycle for an existing file, it does not use the log parameter value although it checks that the value is valid.

PFC\$LOG

Maintain a file access log.

PFC\$NO_LOG

Do not maintain a file access log.

status: VAR of ost\$status;

Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition pfe\$bad_cycle_number
Identifiers pfe\$bad_cycle_option
 pfe\$bad_family_name
 pfe\$bad_local_file_name
 pfe\$bad_log_option
 pfe\$bad_master_catalog_name
 pfe\$bad_nth_subcatalog_name
 pfe\$bad_password
 pfe\$bad_permanent_file_name
 pfe\$bad_retention_period
 pfe\$catalog_full
 pfe\$cycle_overflow
 pfe\$cycle_underflow
 pfe\$duplicate_cycle
 pfe\$incorrect_password
 pfe\$lfn_in_use
 pfe\$name_already_subcatalog
 pfe\$nth_name_not_subcatalog
 pfe\$path_too_short
 pfe\$pf_system_error
 pfe\$unknown_family
 pfe\$unknown_master_catalog
 pfe\$unknown_nth_subcatalog
 pfe\$usage_not_permitted
 pfe\$user_not_permitted

- Remarks**
- If the specified permanent file entry does not exist, PFP\$DEFINE creates the catalog entry for the file and its initial cycle. If the permanent file is already registered in a catalog, PFP\$DEFINE creates a new cycle of the file.
 - At completion of the procedure, the permanent file is attached to the job. During the initial attachment, all access modes are valid, but no sharing of the file is allowed.
 - PFP\$DEFINE defines no access control entries for the file. Therefore, access to the file is initially granted only to users who have access to the catalog to which the file belongs.

PFP\$PURGE

Purpose Removes a permanent file cycle.

NOTE

You must have control permission to the file to purge a file cycle.

Format PFP\$PURGE (path, cycle_selector, password, status)

Parameters **path:** pft\$path;
File path of the file cycle to be purged.

cycle_selector: pft\$cycle_selector;
Permanent file cycle purged.

Field	Content
-------	---------

cycle_option	Key field indicating how the file cycle is specified.
--------------	---

PFC\$LOWEST_CYCLE

Lowest file cycle used.

PFC\$HIGHEST_CYCLE

Highest cycle used.

PFC\$SPECIFIC_CYCLE

Cycle specified by cycle_number field.

cycle_number	Cycle number (integer from 1 through PFC\$MAXIMUM_CYCLE_NUMBER, 999).
--------------	---

password: pft\$password;

File password (1- through 31-character name). If the file has no password, specify a space as the password.

status: VAR of ost\$status;

Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers

- pfe\$bad_cycle_number
- pfe\$bad_cycle_option
- pfe\$bad_family_name
- pfe\$bad_master_catalog_name
- pfe\$bad_nth_subcatalog_name
- pfe\$bad_permanent_file_name
- pfe\$bad_password
- pfe\$incorrect_password
- pfe\$invalid_ring_access
- pfe\$name_not_permanent_file
- pfe\$nth_name_not_subcatalog
- pfe\$path_too_short
- pfe\$pf_system_error
- pfe\$unknown_cycle
- pfe\$unknown_family
- pfe\$unknown_master_catalog
- pfe\$unknown_nth_subcatalog_name
- pfe\$unknown_permanent_file
- pfe\$usage_not_permitted
- pfe\$user_not_permitted

Remarks

- The PFP\$PURGE call releases the space assigned to the cycle. However, if the cycle is attached when the PFP\$PURGE call is issued, NOS/VE does not release the space until all jobs to which the file is attached are terminated. The task that calls PFP\$PURGE continues processing; it is not suspended while the file to be purged remains attached to other jobs.
- If the cycle is the only existing cycle for the file, PFP\$PURGE also removes the catalog entry for the permanent file.
- Removing a file entry also removes all access control entries for the file.
- After the PFP\$PURGE procedure is called for a file cycle, no user can attach that cycle.

Changing File Entry Information

After a file is defined, you can make changes to its file entry information with a call to the PFP\$CHANGE procedure. PFP\$CHANGE can change the following items:

- Permanent file name.
- Password.
- Cycle number.
- Cycle retention period starting from the current date.
- Access log selection.
- Account and project names. The account and project names of the caller become the new account and project names for the file.

PFP\$CHANGE

Purpose Changes information in a permanent file entry.

Format **PFP\$CHANGE (path, cycle_selector, password, change_list, status)**

Parameters **path:** pft\$path;
File path specifying the file entry to be changed.

cycle_selector: pft\$cycle_selector;
Permanent file cycle.

Field	Content
cycle_option	Key field indicating how the file cycle is specified.
	PFC\$LOWEST_CYCLE Lowest cycle used.
	PFC\$HIGHEST_CYCLE Highest cycle used.
	PFC\$SPECIFIC_CYCLE Cycle specified by cycle_number field.
cycle_number	Cycle number (integer from 1 through PFC\$MAXIMUM_CYCLE_NUMBER, 999).

password: pft\$password;
Current file password (1- through 31-character name). If the file does not have a password, specify a space as the password.

change_list: pft\$change_list;
List of catalog entry changes. The list is an adaptable array of PFT\$CHANGE_DESCRIPTOR records (see table 3-1).

status: VAR of ost\$status;
Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers

pfe\$bad_change_type
 pfe\$bad_cycle_number
 pfe\$bad_cycle_option
 pfe\$bad_family_name
 pfe\$bad_log_option
 pfe\$bad_master_catalog_name
 pfe\$bad_nth_subcatalog_name
 pfe\$bad_password
 pfe\$bad_permanent_file_name
 pfe\$bad_retention_period
 pfe\$catalog_full
 pfe\$duplicate_cycle
 pfe\$incorrect_password
 pfe\$name_already_permanent_file
 pfe\$name_already_subcatalog
 pfe\$name_not_permanent_file
 pfe\$nth_name_not_subcatalog
 pfe\$path_too_short
 pfe\$pf_system_error
 pfe\$unknown_cycle
 pfe\$unknown_family
 pfe\$unknown_master_catalog
 pfe\$unknown_nth_subcatalog
 pfe\$unknown_permanent_file
 pfe\$usage_not_permitted
 pfe\$user_not_permitted

- Remarks**
- You must have control permission to the file to change the file entry.
 - You can change the file entry information while the file is attached to another job.

Table 3-1. Change List Record (PFT\$CHANGE_DESCRIPTOR)

Field	Content
change_type	Key field determining the attribute changed (PFT\$CHANGE_TYPE). PFC\$PF_NAME_CHANGE New name in pfn field. PFC\$PASSWORD_CHANGE New password in password field. PFC\$CYCLE_NUMBER_CHANGE New cycle number in cycle_number field. PFC\$RETENTION_CHANGE New retention period in retention field. PFC\$LOG_CHANGE New log option in log field. PFC\$CHARGE_CHANGE The account and project names of the job become the new account and project names for the file.
pfn	New permanent file name (PFT\$NAME, 31 characters).
password	New password (PFT\$PASSWORD, 1- through 31-character name).
cycle_number	New number for the cycle (PFT\$CYCLE_NUMBER, 1 through 999).
retention	New retention period starting from current date (PFT\$RETENTION, 1 through 999 days; 999 specifies infinite retention).
log	New log option (PFT\$LOG, see File Access Log later in this chapter). PFC\$LOG Maintain a file access log. PFC\$NO_LOG Do not maintain a file access log.

File Subcatalogs

A catalog can contain entries defining files as well as entries defining other catalogs. A catalog defined within another catalog is called a subcatalog. Within a file reference, a subcatalog is always preceded by the catalog in which it resides.

Logically, a subcatalog can be named to represent topical headings. File entries having information pertaining to the topic can then be grouped within the structure of the subcatalog. For example, USERX has a subcatalog named PROC which might contain several procedure files.

To define a subcatalog, call the PFP\$DEFINE_CATALOG procedure. To delete a subcatalog, call the PFP\$PURGE_CATALOG procedure.

PF\$DEFINE_CATALOG

Purpose Defines a subcatalog.

NOTE

You must own the catalog in which you define a subcatalog.

Format PF\$DEFINE_CATALOG (path, status)

Parameters **path:** pft\$path;
 Catalog path. The last name in the path list is that of the new subcatalog.

status: VAR of ost\$status;
 Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers pfe\$bad_family_name
 pfe\$bad_last_subcatalog_name
 pfe\$bad_master_catalog_name
 pfe\$bad_nth_subcatalog_name
 pfe\$name_already_permanent_file
 pfe\$name_already_subcatalog
 pfe\$not_master_catalog_owner
 pfe\$nth_name_not_subcatalog
 pfe\$path_too_short
 pfe\$pf_system_error
 pfe\$unknown_family
 pfe\$unknown_master_catalog
 pfe\$unknown_nth_subcatalog

Remarks

- After a subcatalog is defined, files or other subcatalogs can be defined within the subcatalog. Referencing the file or subcatalog requires that you specify each catalog in the catalog path.
- The PF\$DEFINE_CATALOG procedure cannot define a master catalog. Only the family administrator can define a master catalog.

PFP\$PURGE_CATALOG

Purpose Removes a subcatalog.

NOTE

You must own the catalog from which you remove a subcatalog.

Format PFP\$PURGE_CATALOG (path, status)

Parameters **path:** pft\$path;
Catalog path. The last name in the path list is that of the subcatalog to be purged.

status: VAR of ost\$status;
Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers pfe\$bad_family_name
pfe\$bad_last_subcatalog_name
pfe\$bad_master_catalog_name
pfe\$bad_nth_subcatalog_name
pfe\$catalog_not_empty
pfe\$last_name_not_subcatalog
pfe\$not_master_catalog_owner
pfe\$nth_name_not_subcatalog
pfe\$path_too_short
pfe\$pf_system_error
pfe\$unknown_family
pfe\$unknown_last_subcatalog
pfe\$unknown_master_catalog
pfe\$unknown_nth_subcatalog

Remarks

- The subcatalog must be empty before it can be purged. All file and subcatalog entries in the subcatalog must first be purged before removal of the subcatalog can be accomplished.
- The PFP\$PURGE_CATALOG procedure cannot purge a master catalog. Only the family administrator can purge a master catalog.

Access Modes

Access modes protect files in that they allow the file owner to specify the modes of operations that can be performed on the file. A task is granted access to a file only when its requested operation is within a set of permitted access modes for the file. The access modes are listed as follows:

- **Read:** Allows the task to read data from the file.
- **Shorten:** Allows the task to reduce the file length, discarding data existing beyond that length.
- **Append:** Allows the task to add data to the end of the file, lengthening the file.
- **Modify:** Allows the task to change existing data within the file.
- **Execute:** Allows the task to execute the file, assuming the file contains executable object modules.
- **Cycle:** Allows the task to create a new cycle of the file or a new file.
- **Control:** Allows the task to change information in the file entry and purge a file cycle or file.

To perform all possible write operations on a file, the task must have shorten, append, and modify permissions to the file.

File Access Log

When defining a file, you can request that the system maintain a record of the users that access the file. This record is called a file access log. To request this service, you specify PFC\$LOG as the log option parameter on the PFP\$DEFINE call that defines the file.

When you specify PFC\$LOG, the system maintains an access log for each user that accesses the file. Each access log contains the following information:

- The user that accessed the file.
- The number of accesses by the user.
- The date and time of the last access by the user and the last cycle accessed.

You can display the file access log using the DISPLAY_CATALOG_ENTRY command described in the SCL System Interface manual.

Access Control Entries

An access control entry is a set of permitted operations that can be performed on a file or catalog entry at a given time. Access control entries apply either to a specific file entry or to a catalog entry and can only be defined by the owner of the respective file or catalog. The permit selections and share requirements parameters listed on the PFP\$PERMIT call define the access control entry for a file. These same parameters listed on a call to the PFP\$PERMIT_CATALOG procedure define an access control entry for all entries in a catalog.

A file can have several access control entries. Each entry can specify different permit selections for different groups of users. Listed in each access control entry is the specified group of users to which the defined permit selections apply. The following are the user groups to which an individual access control entry could apply:

- All users.
- All users in a family.
- One user in a family.
- All users executing under an account name.
- One user executing under an account name.
- All users executing under an account name and a project name.
- One user executing under an account name and a project name.

An access control entry contains a permit selections set and a share requirements set.

Permit Selections

A file's permit selections set contains the access modes that are valid as usage selections on an attach request. The permit selections set validates the access modes specified on an attach request. When attempting to attach a file with an ATTACH_FILE command or PFP\$ATTACH call, you specify the usage selections for the attach. The usage selections are the access modes in which the task intends to use the file while it is attached.

When validating an attach request, the applicable access control entry compares the usage modes specified on the attach request with the access modes defined in the file's permit selection set. All usage modes specified on the attach must be within the set of access modes defined in the applicable access control entry. If not, the attach attempt fails.

For example, if the permit selections set contains only read access, the access control entry allows only read access to the file. If the permit selections set contains no access permissions, the access control entry allows no access to the file.

If an attach request specifies read and append access as the usage selections on the attach and the permit selections set of the applicable access control entry does not include both read access and append access, the attach attempt fails.

Share Requirements

A file's share requirements set is defined as the share modes in which a job must access the file while it is attached. When attempting to attach a file with an ATTACH_FILE command or PFP\$ATTACH call, you specify the share selections for the attach. The share requirements set validates the share selections specified on an attach request.

The share requirements set contains the minimum set of access modes that are required as share selections on an attach request. All access modes defined in the file's share requirements set must be listed as share selections on the attach request, or the attach attempt fails. For example, if an attach attempt specifies only read access as its share selections set and the share requirements set of the applicable access control entry contains read and append access, the attach attempt fails.

If the share requirements set contains read access permission, the access control entry requires that read access be specified as a share selection on the attach request. If the share requirements set contains no access permissions, you need not specify any share selections on your attach request as the access control entry does not require any; in this case, the attach request is granted exclusive access to the file, not allowing any concurrent attaches.

Multiple Access Control Entries

When you attempt to attach a file that has more than one access control entry that could apply to you, the system determines the applicable access control entry using the following rules:

- If you belong to more than one group for which an access control entry is defined, the access control entry applicable to the smaller group applies.

For example, if one entry applies to all users and another entry applies to a family of users and you belong to that family, the system uses the entry applying to the family of users.

- If access control entries for the same group exist for more than one element of the file path, the entry applicable to the last element in the file path applies.

For example, if a catalog has an access control entry for all users and a file defined within the catalog also has an access control entry for all users, the access control entry defined for the file will be used to validate attach requests.

If necessary, the system uses both rules to determine the applicable access control entry. For example, assume that the following access control entries exist for a file named `FILE_1` in a catalog named `CATALOG_1`:

1. An entry applicable to `CATALOG_1` for the family of users `FAMILY_A`.
2. An entry applicable to `FILE_1` for all users.
3. An entry applicable to `FILE_1` for the family of users `FAMILY_A`.

Assume that the user attempting to attach the file belongs to `FAMILY_A` for which access control entries are defined. Using rule 1, the system determines that a family of users is a smaller group than all users. Because multiple entries are defined for `FAMILY_A`, the system must use rule 2 to determine the applicable access control entry. Using rule 2, `FILE_1` is later in the file path than `CATALOG_1`. Therefore, the entry for `FAMILY_A` defined for `FILE_1` is the access control entry applicable to the attach attempt.

PFP\$PERMIT

Purpose Defines or changes an access control entry for a file.

NOTE

Only the file owner can define an access control entry for a file.

Format **PFP\$PERMIT (path, group, permit_selections, share_requirements, application_info, status)**

Parameters **path:** pft\$path;
File path specifying the file to which the access control entry applies.

group: pft\$group;
User group to which the access control entry applies (variant record of type PFT\$GROUP as described in table 3-2).

permit_selections: pft\$permit_selections;
Set of access permissions granted by the access control entry. A null set indicates that the user group is to be denied access to the file.

PFC\$READ

Read permission.

PFC\$SHORTEN

Shorten permission.

PFC\$APPEND

Append permission.

PFC\$MODIFY

Modify permission.

PFC\$EXECUTE

Execute permission.

PFC\$CYCLE

Cycle permission (permission to create additional file cycles).

PFC\$CONTROL

Control permission.

share_requirements: pft\$share_requirements;

The set of access modes that an attempt to attach the file must specify as share selections. A null set indicates that an attach request may specify no share selections; the attach request could be exclusive, preventing other users from attaching the file at the same time.

PFC\$READ

Read sharing required.

PFC\$SHORTEN

Shorten sharing required.

PFC\$APPEND

Append sharing required.

PFC\$MODIFY

Modify sharing required.

PFC\$EXECUTE

Execute sharing required.

application_info: pft\$application_info;

Additional access information that can be used by application programs (31-character string).

status: VAR of ost\$status;

Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers

pfe\$bad_account_name

pfe\$bad_family_name

pfe\$bad_group_type

pfe\$bad_master_catalog_name

pfe\$bad_nth_subcatalog_name

pfe\$bad_permanent_file_name

pfe\$bad_project_name

pfe\$bad_user_name

pfe\$catalog_full

pfe\$name_not_permanent_file

pfe\$not_master_catalog_owner

pfe\$nth_name_not_subcatalog

pfe\$path_too_short

pfe\$pf_system_error

pfe\$unknown_family

pfe\$unknown_master_catalog

pfe\$unknown_nth_subcatalog

pfe\$unknown_permanent_file

Remarks When replacing an access control entry, be certain to specify the correct group as the group parameter on the PFP\$PERMIT call. The parameters specified as permit_selections, share_requirements, and application_info parameters listed on the call will replace any of those same existing parameters for the particular group of users.

Table 3-2. User Group Record (Type PFT\$GROUP)

Field	Content
group_type	Key field indicating the group type. PFC\$PUBLIC All users. PFC\$FAMILY User family specified in the family_description field. PFC\$ACCOUNT Account specified in the account_description field. PFC\$PROJECT Project specified in the project_description field. PFC\$USER User specified in the user_description field. PFC\$USER_ACCOUNT User specified in the user_account_description field. PFC\$MEMBER User specified in the member_description field.
family_description	Record containing the following field: family Family name (type OST\$FAMILY_NAME).
account_description	Record containing the following fields: family Family name (type OST\$FAMILY_NAME). account Account name (type AVT\$ACCOUNT_NAME).

(Continued)

Table 3-2. User Group Record (Type PFT\$GROUP) (Continued)

Field	Content
project_description	Record containing the following fields: family Family name (type OST\$FAMILY_NAME). account Account name (type AVT\$ACCOUNT_NAME). project Project name (type AVT\$PROJECT_NAME).
user_description	Record containing the following fields: family Family name (type OST\$FAMILY_NAME). user User name (type OST\$USER_NAME).
user_account_description	Record containing the following fields: family Family name (type OST\$FAMILY_NAME). account Account name (type AVT\$ACCOUNT_NAME). user User name (type OST\$USER_NAME).
member_description	Record containing the following fields: family Family name (type OST\$FAMILY_NAME). account Account name (type AVT\$ACCOUNT_NAME). project Project name (type AVT\$PROJECT_NAME). user User name (type OST\$USER_NAME).

PFP\$DELETE_PERMIT

Purpose Removes an access control entry for a file.

NOTE

Only the file owner can delete an access control entry for a file.

Format **PFP\$DELETE_PERMIT (path, group, status)**

Parameters **path:** pft\$path;
File path specifying the file to which the access control entry applies.

group: pft\$group;
User group to which the access control entry applies (variant record of type PFT\$GROUP described in table 3-2).

status: VAR of ost\$status;
Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers

- pfe\$bad_account_name
- pfe\$bad_family_name
- pfe\$bad_group_type
- pfe\$bad_master_catalog_name
- pfe\$bad_nth_subcatalog_name
- pfe\$bad_permanent_file_name
- pfe\$bad_project_name
- pfe\$bad_user_name
- pfe\$name_not_permanent_file
- pfe\$not_master_catalog_owner
- pfe\$nth_name_not_subcatalog
- pfe\$path_too_short
- pfe\$pf_system_error
- pfe\$unknown_family
- pfe\$unknown_master_catalog
- pfe\$unknown_nth_subcatalog
- pfe\$unknown_permanent_file

PFP\$PERMIT_CATALOG

Purpose Defines an access control entry that applies to all files and subcatalogs defined in a catalog.

NOTE

Only the owner of the catalog can define an access control entry for a catalog.

Format **PFP\$PERMIT_CATALOG (path, group, permit_selections, share_requirements, application_info, status)**

Parameters **path:** pft\$path;

Catalog path specifying the catalog to which the access control entry applies. The last name in the path list must be that of the catalog for which the access control entry is defined.

group: pft\$group;

User group to which the access control entry applies (variant record of type PFT\$GROUP as described in table 3-2).

permit_selections: pft\$permit_selections;

Set of access permissions granted by the access control entry. A null set indicates that the user group is to be denied access to all files in the catalog unless granted access by an access control entry for the file.

PFC\$READ

Read permission.

PFC\$SHORTEN

Shorten permission.

PFC\$APPEND

Append permission.

PFC\$MODIFY

Modify permission.

PFC\$EXECUTE

Execute permission.

PFC\$CYCLE

Cycle permission (grants permission to create new entries in the catalog).

PFC\$CONTROL

Control permission.

share_requirements: pft\$share_requirements;

The set of access permissions that the attach request must specify as share selections. A null set indicates that the attach request may specify no share selections; the attach request could be exclusive, preventing other users from attaching the file at the same time.

PFC\$READ

Read sharing required.

PFC\$SHORTEN

Shorten sharing required.

PFC\$APPEND

Append sharing required.

PFC\$MODIFY

Modify sharing required.

PFC\$EXECUTE

Execute sharing required.

application_info: pft\$application_info;

Additional access information that can be used by application programs (31-character string).

status: VAR of ost\$status;

Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers

pfe\$bad_account_name
 pfe\$bad_family_name
 pfe\$bad_group_type
 pfe\$bad_last_subcatalog_name
 pfe\$bad_master_catalog_name
 pfe\$bad_nth_subcatalog_name
 pfe\$bad_project_name
 pfe\$bad_user_name
 pfe\$catalog_full
 pfe\$last_name_not_subcatalog
 pfe\$nth_name_not_subcatalog
 pfe\$not_master_catalog_owner
 pfe\$path_too_short
 pfe\$pf_system_error
 pfe\$unknown_family
 pfe\$unknown_last_subcatalog
 pfe\$unknown_master_catalog
 pfe\$unknown_nth_subcatalog

Remarks

- The access control entry created validates access by a group or groups of users to all files and subcatalogs registered in the catalog specified on the PFP\$PERMIT_CATALOG call.
- When replacing an access control entry for a catalog or subcatalog, be certain to specify the correct group of users on the group parameter for the PFP\$PERMIT_CATALOG call. The parameters specified as permit_selections, share_requirements, and application_info listed on the call will replace any of those same existing parameters for the particular group of users.

PFP\$DELETE_CATALOG_PERMIT

Purpose Removes an access control entry that applies to a catalog.

NOTE

Only the catalog owner can delete an access control entry for a catalog.

Format **PFP\$DELETE_CATALOG_PERMIT (path, group, status)**

Parameters **path:** pft\$path;
Catalog path specifying the catalog to which the access control entry applies. The last name in the path list is that of the subcatalog.

group: pft\$group;
User group to which the access control entry applies (variant record of type PFT\$GROUP as described in table 3-2).

status: VAR of ost\$status;
Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

Condition Identifiers pfe\$bad_account_name
pfe\$bad_family_name
pfe\$bad_group_type
pfe\$bad_last_subcatalog_name
pfe\$bad_master_catalog_name
pfe\$bad_nth_subcatalog_name
pfe\$bad_permanent_file_name
pfe\$bad_project_name
pfe\$bad_user_name
pfe\$last_name_not_subcatalog
pfe\$not_master_catalog_owner
pfe\$nth_name_not_subcatalog
pfe\$path_too_short
pfe\$pf_system_error
pfe\$unknown_family
pfe\$unknown_last_subcatalog
pfe\$unknown_master_catalog
pfe\$unknown_nth_subcatalog

Attaching a Permanent File

You can attach a permanent mass storage file to your job using an `ATTACH_FILE` command, a `PFPS$ATTACH` call, or by using a file reference which specifies the file path, and may also specify the file cycle, and the file open position. An `ATTACH_FILE` command or `PFPS$ATTACH` call can offer a task greater selectivity in using the file than can a file reference. Attach commands or calls can specify a password for the file and can select the option of waiting for a file that is currently attached. These options are not available to the user of a file reference.

Attaching a File with `PFPS$ATTACH` or `ATTACH_FILE`

When you make an attach request for a permanent file, the permit selections and share requirements in the file's applicable access control entry will govern whether the attach request will be allowed. The access modes listed as the `usage_selections` parameter on the `PFPS$ATTACH` call must be within the set of access modes defined in the file's `permit_selections`. If any share modes, as listed in the file's `share_requirements` set were defined for the file, these same share modes must be included in the `share_selections` parameter on the attach request.

Upon successful completion of the `PFPS$ATTACH` call or `ATTACH_FILE` command, the file becomes scheduled within the job for those modes of access specified on the attach. Subsequent access to the file while it is attached must be a subset of these same modes of access. You can examine the `global_access_mode` and `global_share_mode` attributes returned by a `DISPLAY_FILE_ATTRIBUTES` command or an `AMP$GET_FILE_ATTRIBUTES` call to determine the respective `usage_selections` and `share_selections` specified on the most recent successful attach. The `global_access_mode` and `global_share_mode` attributes as well as the `AMP$GET_FILE_ATTRIBUTES` call is defined in chapter 6 of this manual. The `DISPLAY_FILE_ATTRIBUTES` command is defined in the SCL System Interface manual.

Attaching a File with a File Reference

You can reference a file for use within a job by specifying its full or relative path name. Such a reference is termed a file reference. A permanent file reference results in the file being implicitly scheduled when it is initially opened within the job. The usage_selections for which the file will be scheduled within the job are determined by the access_modes specified on the AMP\$OPEN call that opens the file. If no access_modes are explicitly specified when the AMP\$OPEN call occurs, the usage_selections will default to those modes of access specified in the file's permit_selections set in the file's applicable access control entry. The applicable access control entry is qualified by the ring of the caller of AMP\$OPEN.

Since the AMP\$OPEN call does not have available to it the ability to specify a share_selections set when it opens a file, NOS/VE examines the file's access control entry and determines the modes of sharing that will be allowed for the file within the job. If file's access control entry has any kind of write access specified in its permit_selections, NOS/VE sets the file's share_modes to the null set; that is, no sharing of the file will be allowed while it is attached. If access_modes other than write are contained in the file's permit_selections, the share_modes set for the file will be set to PFC\$READ and PFC\$EXECUTE.

When you schedule a permanent file cycle within a job for some form of write access, and you want other jobs to be able to share the file while it is attached, you must use either the attach call or command and specify the share_modes in which you are willing to share the file prior to opening it.

Evaluating Attach Requests

A user can attach a file only if an applicable access control entry exists for the user. The system determines the applicable access control entry as described earlier in this chapter under Multiple Access Control Entries.

Having found the user's applicable access control entry, the system validates the requested attach as follows:

- Each usage mode specified on the attach request must be within the access modes defined in the file's permit selections set.
- The share selections set specified on the attach request must include all share modes defined in the file's share requirements set.

When a new attach request is issued for a file cycle currently attached to a job, the following compatibility checks are made:

- The usage selections specified on a new attach request must be within the share selection set of any current attaches.
- The share selections set, as specified on the new attach request, must include all usage selections of any current attaches.

File Cycle Busy Status

A PFP\$ATTACH call returns the abnormal status PFE\$CYCLE_BUSY if the requested file cycle is busy. A file cycle is busy if the attach request specifies a usage selections set or share selections set that is incompatible with the current attaches of the file as outlined above.

An example which shows the interaction of several attach attempts for the same file cycle is presented on the next page.

For example, the following is a file sharing example that consists of a sequence of several attach requests for the same file cycle. Assume that the first attach attempt has been granted access; therefore, the parameters specified on its attach request were within the permit selections and share requirements as defined in the file's applicable access control entry. Also assume the all the requests in the sequence are governed by the same access control entry.

In the following example, the entire sequence of attach attempts occurs before any of the successful attaches returns the file. The successful attach attempts 1 and 4 specified share selections on their attach requests that restrict any subsequent attaches.

Attach Attempt	Usage Selections	Share Selections	Result
1	Read	Read Execute	Normal status. The file cycle is not currently attached, so no compatibility check is required.
2	Append	Read Execute	PFE\$CYCLE_BUSY status. Append was not specified as a share selection in attach 1.
3	Execute	Execute	PFE\$CYCLE_BUSY status. The attach request's share selection does not include the usage selections specified in attach 1.
4	Read	Read	Normal status. The attach attempt requests read access that is in the share selections set of attach 1. The specified share selections set includes the read access as specified in the usage selections of attach 1.
5	Execute	Read	PFE\$CYCLE_BUSY status. The attach attempt requests execute access that is in the share selections set of attach 1 but is not in the share selections set of attach 4.

Wait Option

The wait option on the attach request determines whether (if the file cycle is busy) the task waits for the file cycle or returns the abnormal status PFE\$CYCLE_BUSY. If the attach request specifies the wait option and the file cycle is busy, other attach requests for the file cycle can be processed while the task waits for the file cycle.

For example, suppose USER1 attaches a file cycle with share selections read and append. USER2 attempts to attach the file cycle with usage selection execute. If USER2 requested the wait option, his or her task is suspended until USER1 returns the file cycle. Suppose that, while USER2 is waiting for the file cycle, USER3 also attaches the file cycle with share selections read and append. Now, USER2 must wait until both USER1 and USER3 have returned the file cycle.

PFP\$ATTACH

Purpose Explicitly attaches a permanent file cycle to a job.

Format **PFP\$ATTACH (lfn, path, cycle_selector, password, usage_selections, share_selections, wait, status)**

Parameters **lfn:** amt\$local_file_name;
Local file name.

path: pft\$path;
File path identifying the file to be attached.

cycle_selector: pft\$cycle_selector;
Permanent file cycle.

Field	Content
cycle_option	Key field indicating how the file cycle is specified.
	PFC\$LOWEST_CYCLE Lowest file cycle used.
	PFC\$HIGHEST_CYCLE Highest cycle used.
	PFC\$SPECIFIC_CYCLE Cycle specified by cycle_number field.
cycle_number	Cycle number (integer from 1 through PFC\$MAXIMUM_CYCLE_NUMBER, 999).

password: pft\$password;
File password (1- through 31-character name). If the file has no password, specify a space for the password.

usage_selections: pft\$usage_selections;

Set of access modes that the job requires. The usage_selections set limits the access modes specified on calls to open the file while it is attached.

PFC\$READ

The job can read the file.

PFC\$SHORTEN

The job can shorten the file.

PFC\$APPEND

The job can append data to the file, thereby lengthening it.

PFC\$MODIFY

The job can modify data within the file.

PFC\$EXECUTE

The job can execute the file.

share_selections: pft\$share_selections;

Set of access modes that subsequent attempts to attach the file cycle can specify as usage selections. The share_selections set must include all access modes in the share_requirements set in the applicable access control entry; it can also include additional access modes not included in the share_requirements set.

PFC\$READ

The file can be attached for read access.

PFC\$SHORTEN

The file can be attached for shorten access.

PFC\$APPEND

The file can be attached for append access.

PFC\$MODIFY

The file can be attached for modify access.

PFC\$EXECUTE

The file can be attached for execute access.

wait: pft\$wait;

Action if the file cycle is busy.

PFC\$WAIT

PFP\$ATTACH waits until the file is available and then attaches the file.

PFC\$NOWAIT

PFP\$ATTACH completes without attaching the file; it returns the PFE\$CYCLE_BUSY condition code in the status record.

status: VAR of ost\$status;

Status variable. The product identifier returned is PFC\$PERMANENT_FILE_MANAGER_ID.

**Condition
Identifiers**

pfe\$bad_cycle_number
 pfe\$bad_cycle_option
 pfe\$bad_family_name
 pfe\$bad_local_file_name
 pfe\$bad_master_catalog_name
 pfe\$bad_nth_subcatalog_name
 pfe\$bad_password
 pfe\$bad_permanent_file_name
 pfe\$bad_wait_option
 pfe\$catalog_full
 pfe\$cycle_busy
 pfe\$incorrect_password
 pfe\$invalid_ring_access
 pfe\$lfn_in_use
 pfe\$name_not_permanent_file
 pfe\$nth_name_not_subcatalog
 pfe\$path_too_short
 pfe\$pf_system_error
 pfe\$sharing_not_permitted
 pfe\$undefined_data
 pfe\$unknown_cycle
 pfe\$unknown_family
 pfe\$unknown_master_catalog
 pfe\$unknown_nth_subcatalog
 pfe\$unknown_permanent_file
 pfe\$usage_not_permitted
 pfe\$user_not_permitted

Remarks

- If the permanent file cycle is already attached to the job, a task within the job need not attach the file cycle before processing it. (This assumes that the local file name associated with the instance of attach is known to the task).
- A task can attach an already attached file cycle again using a different local file name. However, the file cycle cannot be attached with usage selections or share selections that conflict with other current attaches of the file cycle. Procedure calls within the task could reference the attached file cycle by either local file name. After task completion, the file cycle remains attached to the job under both local file names. To detach the file cycle before the job terminates, you must issue an AMP\$RETURN call or DETACH_FILE command for each local file name.



Tape Management

Tape File Requests	4-1
Multivolume Tape Files	4-2
RMP\$REQUEST_TAPE	4-3
Tape File Attributes	4-6
Tape File Positioning	4-7
Open Positioning	4-7
Close Positioning	4-7
Rewind Positioning	4-7
Skip Positioning	4-7
Forward Skip by Tapemarks	4-8
Backward Skip by Tapemarks	4-8
AMP\$SKIP_TAPE_MARKS	4-9
Embedded Tapemarks	4-11
Copying Tape Files	4-11
AMP\$WRITE_TAPE_MARK	4-12



NOS/VE supports unlabeled 9-track tape files. The file structure for unlabeled tape files is indicated by tapemarks. Tape files are defined as the data between two nonconsecutive tapemarks. Two consecutive tapemarks indicate the end of a tape volume.

All tape files have sequential file organization. The description of record access for sequential file organization in chapter 9 applies to tape files.

Unlike the other device classes, tape files cannot have more than one instance of open at a time. An open tape file must be closed before it can be opened again by the same task or another task.

Unlike permanent mass storage files, the file attribute set of a tape file is not preserved with the file data. NOS/VE discards a tape file attribute set after a DETACH_FILE command or AMP\$RETURN call ends the file assignment.

The following procedures described in this chapter perform specific tape functions:

- RMP\$REQUEST_TAPE: Associates a local file name with the magnetic tape device class.
- AMP\$SKIP_TAPE_MARKS: Positions a tape file by skipping forward and backward a specified number of tapemarks.
- AMP\$WRITE_TAPE_MARK: Writes a tapemark on a tape file.

Tape File Requests

The RMP\$REQUEST_TAPE procedure associates a local file name with the magnetic tape device class and provides device specifications to be used if the file is opened.

The device specifications include the following values:

- Tape transport type: 9-track.
- Recording density: 800, 1600, or 6250 cpi (1600 and 6250 cpi densities are recommended due to the inherent reliability of the tape recording technique).
- Write ring requirement.
- Volumes included in the file.

Multivolume Tape Files

NOS/VE supports multivolume tape files. All device specifications for the file, including the write ring requirement, apply to all volumes in the file.

NOS/VE manages volume switching. When it encounters the end of the current volume, it refers to the volume list to determine the next volume in the file. (The end-of-volume indicator for a get call is two consecutive tapemarks; the end-of-volume indicator for a put call is the end-of-tape reflective marker.)

If a subsequent volume exists in the list, the system requests the operator to mount the next volume on an appropriate tape unit. When the operator assigns the volume to a tape unit, the system changes the file assignment to the tape unit on which the next volume is mounted. Only one tape unit is assigned to the file at a time.

If no subsequent volume exists in the volume list, subsequent processing depends on whether the task had issued a get call or a put call. For a get call, AMC\$EOI is returned as the file position, and the call terminates.

When no subsequent volume exists for a put call, the system operator is asked to supply an additional volume. The operator must either supply a tape volume or terminate the job. If the operator supplies an additional volume, the put operation continues.

If a successful volume switch occurs while the task is putting data on the tape file, the system writes an end-of-volume indicator (two consecutive tape marks) and continues the put operation at the beginning of the next volume. The task is not aware of or affected by the volume switch.

RMP\$REQUEST_TAPE

Purpose Associates a local file name with the magnetic tape device class and provides the device specifications used if the file is opened.

Format **RMP\$REQUEST_TAPE (local_file_name, class, density, write_ring, volume_list, status)**

Parameters **local_file_name:** amt\$local_file_name;
Local file name.

class: rmt\$tape_class;
Tape unit type.

RMC\$MT9
Nine-track tape unit.

density: rmt\$density;
Tape recording density.

RMC\$800
800 cpi

RMC\$1600
1600 cpi

RMC\$6250
6250 cpi

The 1600 and 6250 cpi densities are recommended due to the inherent reliability of the tape recording technique.

write_ring: rmt\$write_ring;

Indicates whether or not a write ring must be inserted in each tape volume. A tape unit cannot write on a tape volume unless the tape volume has a write ring.

RMC\$WRITE_RING
A write ring must be inserted.

RMC\$NO_WRITE_RING
No write ring should be inserted.

volume_list: rmt\$volume_list;

List of volume serial numbers (vsn) identifying the tape volumes of the file (adaptable array of type rmt\$volume_descriptor records).

Field	Content
recorded_vsn	This field is currently unused.
external_vsn	Six-character volume serial number visible on the tape canister (type RMT\$EXTERNAL_VSN).

status: VAR of ost\$status;

Status variable. The process identifier returned is RMC\$RESOURCE_MANAGEMENT_ID.

Condition Identifiers

- rme\$improper_class_value
- rme\$improper_density_value
- rme\$improper_external_vsn_value
- rme\$improper_recorded_vsn_value
- rme\$improper_write_ring_value

Remarks

- A tape unit is not assigned to the job until the tape file is opened. If the file is never opened, the file is not assigned to a specific tape device, and its association with a device class has no effect.
- A REQUEST_TERMINAL command can override the device class association specified by an RMP\$REQUEST_TAPE call. Subsequent RMP\$REQUEST_TERMINAL or RMP\$REQUEST_NULL_DEVICE calls can also change the device class association if issued before the file is opened.
- A REQUEST_MAGNETIC_TAPE command supercedes any RMP\$REQUEST_TAPE or program request to assign the file to another device class.
- If the task is to have more than one tape file open at the same time, a RESERVE_RESOURCE command must reserve the required number of tape units before the first tape file is opened.
- When a tape volume is assigned to a job, NOS/VE records the following information in the job log:
 - Name of the tape unit on which the volume is mounted.
 - Whether a write ring is inserted in the mounted volume.

You can display job log information with the SCL command DISPLAY_LOG.

Tape File Attributes

As for any other device class, you can set file attribute values for tape files with the file attribute definition calls described in chapter 6. However, unlike a permanent mass storage file, tape file attribute values are not preserved with the file data. NOS/VE discards the attribute values after a DETACH_FILE command or AMP\$RETURN call ends the file assignment.

The file_ organization attribute for tape files must be sequential. All tape files are written and read sequentially using record access calls described in part III.

If you specify system-specified blocking when writing a tape file, the system may pad the last block of the file with circumflex characters. Because the file attributes (including the file length) are not stored with a tape file, the system does not know the exact length of the file when it reads the file. So it reads the entire last block of the file (including any padding characters) as data. Therefore, the program that reads the tape file must check for and discard circumflex characters at the end of the file.

If you specify user-specified blocking when writing a tape file, the system pads any block shorter than the min_block_length value for the file with circumflex characters. To avoid insertion by the system of circumflex characters into the file data, ensure that the min_block_length value is shorter than the shortest record to be written to the file.

Currently, NOS/VE does not perform character code conversion as the result of the character_conversion and internal_code file attribute values. However, a program can retrieve the file attribute values to determine the conversion the program itself should perform on the file data. The means of setting and retrieving file attribute values is described in chapter 6, Defining File Attributes.

Tape File Positioning

A tape file is positioned in response to close, get, put, rewind, and skip calls.

When a tape file is closed, rewound, or repositioned after put calls have been issued, the system ensures that all data from previous put calls is recorded on the tape and then writes two tapemarks to mark the end of the current volume.

Open Positioning

An `open_position` of `AMC$OPEN_AT_BOI` results in the rewinding and dismounting of any currently mounted volume of tape and the mounting of the first volume of the file from the volume list.

An `open_position` of `AMC$OPEN_NO_POSITIONING` results in the physical position of the tape remaining unchanged. An `open_position` of `AMC$OPEN_AT_EOI` is treated as `AMC$OPEN_NO_POSITIONING`.

Close Positioning

When a tape file is closed after one or more put calls, it is left positioned immediately before the two tapemarks which mark the end of the volume.

Rewind Positioning

When a tape file is rewound, it is positioned at the beginning of the first volume of the file.

Skip Positioning

An `AMP$SKIP` call can reposition the tape by records. An `AMP$SKIP_TAPE_MARKS` call can reposition the tape by tapemarks. (The `AMP$SKIP` call is described in chapter 7, *Opening and Closing Files*; the `AMP$SKIP_TAPE_MARKS` call is described in this chapter.)

Forward Skip by Tapemarks

For a forward skip, the AMP\$SKIP_TAPE_MARKS procedure reads the tape until it has read the specified number of nonconsecutive tapemarks. No tape data is transferred to access method buffers.

If the procedure reads the specified number of nonconsecutive tapemarks, it returns normal status and leaves the file positioned after the last tapemark read.

If the procedure encounters two consecutive tapemarks, neither tapemark is counted. Instead, the double tapemark causes the procedure to switch the file assignment to the next volume of the tape file; it continues the skip operation using the data on the next volume. If the current volume is the last volume of the file, the skip terminates, returning abnormal status.

Backward Skip by Tapemarks

Before skipping backward, AMP\$SKIP_TAPE_MARKS writes to the tape any data written to the file by a previous operation. AMP\$SKIP_TAPE_MARKS then writes two consecutive tapemarks to terminate the volume before skipping backward.

AMP\$SKIP_TAPE_MARKS skips backward until it finds the specified number of tapemarks or the beginning of the volume. No tape data is transferred to access method buffers.

If AMP\$SKIP_TAPE_MARKS reads the specified number of nonconsecutive tapemarks, it returns normal status and leaves the file positioned before the last tapemark read (positioned past the last tapemark counted while skipping toward the beginning of the volume).

If AMP\$SKIP_TAPE_MARKS reads the beginning of the tape volume, it returns abnormal status and leaves the file positioned at the beginning of the volume.

AMP\$SKIP_TAPE_MARKS

Purpose Repositions a tape file forward or backward the specified number of tapemarks.

NOTE

Read permission to the file is required. The file must not be open when the AMP\$SKIP_TAPE_MARKS call is issued.

Format AMP\$SKIP_TAPE_MARKS (*local_file_name*, *direction*, *count*, *status*)

Parameters **local_file_name**: amt\$local_file_name;
Local file name.

direction: amt\$skip_direction;
Direction of skip.

AMC\$FORWARD
Skip forward.

AMC\$BACKWARD
Skip backward.

count: amt\$tape_mark_count;
Number of tapemarks to be skipped (integer from 1 through 40,000).

status: VAR of ost\$status;
Status variable. The process identifier returned is AMC\$ACCESS_METHOD_ID.

Condition Identifiers

- ame\$file_not_closed
- ame\$file_not_known
- ame\$improper_ANSI_operation
- ame\$improper_device_class
- ame\$improper_skip_count
- ame\$improper_skip_direction
- ame\$skip_encountered_bov
- ame\$skip_encountered_eov
- ame\$skip_requires_read_perm
- ame\$uncertain_tape_position

Remarks

- After normal termination of a forward skip, the file is positioned after the last tapemark skipped. After normal termination of a backward skip, the file is positioned before the last tapemark skipped (towards the beginning of the volume).
- The two consecutive tapemarks that indicate the end of a volume are not included in a tapemark count.

Embedded Tapemarks

The AMP\$WRITE_TAPE_MARK procedure can write a tapemark on an unlabeled tape file. It can be used to write a single embedded tapemark to partition data within a tape file.

A program to read a tape containing single tapemarks must be able to distinguish between a single tapemark and a double tapemark. A get call that encounters a tapemark, whether a single tapemark or a double tapemark, returns a file position of AMC\$EOI. The program must call the AMP\$FETCH_ACCESS_INFORMATION procedure to determine the volume position.

If the volume position is AMC\$EOV, the file is positioned at the end of the last volume in the list. If the volume position is AMC\$AFTER_TAPEMARK, the file is positioned after a single tapemark; a subsequent get call reads the next record after the single tapemark.

Writing two consecutive tapemarks indicates the end of the accessible data on the tape volume. Additional data could be written following the writing of two consecutive tapemarks, but NOS/VE cannot read the data, nor can it position the file between two consecutive tapemarks.

Copying Tape Files

Each AMP\$COPY_FILE call copies one file. A tape file is the data between nonconsecutive tapemarks. If a tape volume contains more than one tape file, a separate AMP\$COPY_FILE call with an open_position of AMC\$OPEN_NO_POSITIONING is required to copy each tape file.

The first AMP\$COPY_FILE call copies data up to the first embedded tapemark. The next AMP\$COPY_FILE call begins copying after the first tapemark and continues to the second tapemark. If the file extends past the end of the tape volume, the system automatically switches volumes as described under Multivolume Tape Files. AMP\$COPY_FILE does not terminate when it encounters the two consecutive tapemarks that indicate the end of a tape volume unless the last volume was read.

The last tape file has been copied when an AMP\$COPY_FILE call returns the exception condition AME\$INPUT_FILE_AT_EOI.

AMP\$COPY_FILE does not write embedded tapemarks on the output file. To copy embedded tapemarks as well as file data, the program must open the file with AMC\$OPEN_NO_POSITIONING, call the AMP\$WRITE_TAPE_MARK procedure to write each tapemark, and then close the file.

AMP\$WRITE_TAPE_MARK

Purpose	Writes a tapemark on a tape file.
Format	AMP\$WRITE_TAPE_MARK (file_identifier, status)
Parameters	file_identifier: amt\$file_identifier; File identifier returned by the AMP\$OPEN call that opened the file. status: VAR of ost\$status; Status variable. The process identifier returned is AMC\$ACCESS_METHOD_ID.
Condition Identifiers	ame\$conflicting_access_level ame\$improper_ANSI_operation ame\$improper_device_class ame\$improper_output_attempt ame\$ring_validation_error ame\$unrecovered_write_error
Remarks	<ul style="list-style-type: none">• Any blocks in memory are written before the tapemark. The call terminates the current block.• The call is invalid for mass storage files and files opened for segment access.

Terminal Management

- Default Terminal Attributes 5-1
 - IFP\$TERMINAL 5-2
 - IFP\$GET_DFLT_TERM_ATTRIBUTES 5-4
- Terminal File Requests 5-5
 - RMP\$REQUEST_TERMINAL 5-6
 - IFP\$GET_TERMINAL_ATTRIBUTES 5-7
- Changing Terminal Attribute Values After the File Is Open 5-8
 - IFP\$STORE_TERMINAL 5-9
 - IFP\$FETCH_TERMINAL 5-11
- Terminal Attributes 5-12
- Special Considerations for Terminal File Processing 5-26
 - File Attributes 5-26
 - File Access Information 5-26
 - File Interface Calls 5-27
 - Terminal Input 5-28
 - Typed Ahead Input 5-29
- Terminal Output 5-30
 - Format Effectors 5-30
 - Logical Lines 5-31
 - Page Wait 5-32
 - Line Folding 5-32
- Terminal Conditions 5-33



This chapter describes the calls that perform the following functions:

- Associate a local file name with the interactive terminal device class.
- Change and retrieve terminal attribute values.

Each interactive job has one and only one interactive terminal associated with it. Each file belonging to the interactive terminal device class within the job is associated with the job's terminal. A request to read data from the file reads data input at the terminal; a request to write data to the file displays data at the terminal.

Default Terminal Attributes

To perform interactive I/O, NOS/VE communicates with the Network Access Method (NAM). NAM validates you, the interactive user, before you can log in to NOS/VE. When you log in to NOS/VE, NAM passes the initial set of default terminal attributes for the job to NOS/VE.

NOTE

A NOS/VE user should not use NAM commands to change terminal attributes. You should use only NOS/VE commands and calls to change terminal attributes.

After logging in to NOS/VE, you can change your default terminal attributes with the SCL command `SET_TERMINAL_ATTRIBUTES`.

When you execute a task, NOS/VE initially assigns the task the default terminal attributes of the job (including the values specified by `SET_TERMINAL_ATTRIBUTES` commands). A task can change its default terminal attribute values by using an `IFP$TERMINAL` call; however, a value specified on an `IFP$TERMINAL` call is effective only if it was not previously set by a `SET_TERMINAL_ATTRIBUTES` command.

A task can retrieve the current values of its default terminal attributes by calling `IFP$GET_DFLT_TERM_ATTRIBUTES`. The call also returns the source of each attribute.

IFP\$TERMINAL

Purpose Sets terminal default values.

NOTE

An IFP\$TERMINAL call cannot override a value set by a previous SET_TERMINAL_ATTRIBUTES command. An attribute value specified by an IFP\$TERMINAL call is effective only if it has not been set by a previous SET_TERMINAL_ATTRIBUTES command.

Format IFP\$TERMINAL (attributes, status)

Parameters **attributes:** ift\$terminal_request_attributes;

Terminal attributes (type IFT\$TERMINAL_REQUEST_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier and attribute value for each record.

The call cannot change the following attributes:

```

abort_line_char
backspace_char
cancel_line_char
network_control_char
output_flow-control
parity
pause_break_char
terminal_class
terminate_break_char

```

status: VAR of ost\$status;

Status variable.

**Condition
Identifiers**

ife\$auto_input_mode_range
 ife\$cr_idle_range
 ife\$current_job_not_interactive
 ife\$echoplex_range
 ife\$lf_idle_range
 ife\$no_format_effectors_range
 ife\$no_transp_delim_selection
 ife\$page_length_range
 ife\$page_wait_range
 ife\$page_width_range
 ife\$prompt_file_name_ill_formed
 ife\$prompt_file_name_not_found
 ife\$prompt_file_name_not_term
 ife\$prompt_string_size_range
 ife\$special_editing_range
 ife\$transp_count_select_range
 ife\$transp_delim_count_range
 ife\$transp_timeout_select_range
 ife\$transparent_mode_range
 ife\$unknown_attribute_key
 ife\$unknown_input_device
 ife\$unknown_output_device
 ife\$unknown_parity_mode
 ife\$unknown_store_attr_key
 ife\$unknown_terminal_class

Remarks

The default values established by the call apply to all files that the task subsequently associates with the interactive terminal class.

IFP\$GET_DFLT_TERM_ATTRIBUTES

Purpose	Returns the current default terminal attribute values for the task.
Format	IFP\$GET_DFLT_TERM_ATTRIBUTES (attributes, status)
Parameters	<p>attributes: VAR of ift\$get_terminal_attributes; Terminal attributes (type IFT\$GET_TERMINAL_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier for each record; the procedure returns the attribute source and value in the record. The attribute sources are listed in table 5-1.</p> <p>status: VAR of ost\$status; Status variable.</p>
Condition Identifiers	<p>ife\$current_job_not_interactive ife\$unknown_attribute_key</p>
Remarks	IFP\$GET_DFLT_TERM_ATTRIBUTES returns the attribute source with the attribute value. The attribute sources are listed in table 5-1.

Table 5-1. Terminal Attribute Sources

Constant Identifier	Attribute Value Source
IFC\$UNDEFINED_ATTRIBUTE	No attribute value.
IFC\$NAM_DEFAULT	NAM default value.
IFC\$OS_DEFAULT	NOS/VE default value.
IFC\$TERMINAL_COMMAND	SET_TERMINAL_ATTRIBUTE command.
IFC\$TERMINAL_REQUEST	IFP\$TERMINAL call.
IFC\$REQUEST_TERMINAL_REQUEST	RMP\$REQUEST_TERMINAL call.
IFC\$STORE_TERMINAL_REQUEST	IFP\$STORE_TERMINAL call.
IFC\$BAM_REQUEST	File attribute definition call.

Terminal File Requests

The RMP\$REQUEST_TERMINAL procedure associates a local file name with the interactive terminal device class and with a terminal attribute set.

The terminal attribute set consists of the default attribute set for the task and any attributes specified on the RMP\$REQUEST_TERMINAL call. A value specified on the RMP\$REQUEST_TERMINAL call overrides all values previously specified for the attribute, including values specified by a SET_TERMINAL_ATTRIBUTES command or IFP\$TERMINAL call.

NOTE

When writing a program that sets terminal attributes, you should determine which attributes the user of the program should be allowed to override using SET_TERMINAL_ATTRIBUTE commands. Attributes that the user may override with SET_TERMINAL_ATTRIBUTES commands are specified on IFP\$TERMINAL calls; attributes that the user may not override are specified on the RMP\$REQUEST_TERMINAL call for the file.

The task can retrieve the terminal attribute values associated with a local file name by calling IFP\$GET_TERMINAL_ATTRIBUTES. The call also returns the source of each attribute.

RMP\$REQUEST_TERMINAL

Purpose	Associates a file with the interactive terminal device class and specifies its terminal attributes.
Format	RMP\$REQUEST_TERMINAL (local_file_name , attributes , status)
Parameters	<p>local_file_name: amt\$local_file_name; Local file name.</p> <p>attributes: ift\$req_terminal_req_attributes; Terminal attributes (type IFT\$REQ_TERMINAL_REQ_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier and attribute value for each record. The call cannot change the values of the following attributes.</p> <p>abort_line_char backspace_char cancel_line_char network_control_char output_flow_control page_length page_width parity pause_break_character terminal_class terminate_break_character</p> <p>status: VAR of ost\$status; Status variable.</p>
Condition Identifiers	<p>rme\$improper_term_attr_key rme\$improper_term_attr_value</p>
Remarks	A value specified on the RMP\$REQUEST_TERMINAL call overrides all values previously specified for the attribute including values specified by a SET_TERMINAL_ATTRIBUTES command. If the file specified on the command is never opened, the file is not assigned to a terminal device, and its association with a device class has no effect (unless it was created by a CREATE_FILE command).

IFP\$GET_TERMINAL_ATTRIBUTES

Purpose Returns terminal attribute values before the file is opened.

Format IFP\$GET_TERMINAL_ATTRIBUTES (*local_file_name*, *attributes*, *status*)

Parameters **local_file_name**: amt\$local_file_name;
Local file name.

attributes: VAR of ift\$get_terminal_attributes;
Terminal attributes (type IFT\$GET_TERMINAL_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier for each record; the procedure returns the attribute source and value in the record. The attribute sources are listed in table 5-1.

status: VAR of ost\$status;
Status variable.

Condition Identifiers ife\$current_job_not_interactive
ife\$file_name_ill_formed
ife\$file_name_not_terminal
ife\$file_name_not_found
ife\$unknown_attribute_key

Remarks The request returns the terminal attribute values that would be in effect if the file was opened immediately following this request. It enables the task to determine whether the current terminal attribute values are appropriate for the processing to follow.

Changing Terminal Attribute Values After the File Is Open

When a terminal file is opened, its initial set of terminal attributes are those associated with the local file name. After the file is opened, the task can change terminal attribute values with calls to `IFP$STORE_TERMINAL`.

The task can retrieve the terminal attribute values currently in effect for the instance of open by calling `IFP$FETCH_TERMINAL`. The call also returns the source of each attribute.

IFP\$STORE_TERMINAL

- Purpose** Changes terminal attribute values after the file is opened.
- Format** **IFP\$STORE_TERMINAL (file_identifier, attributes, status)**
- Parameters** **file_identifier:** amt\$file_identifier;
File identifier returned when the file is opened.
- attributes:** ift\$store_terminal_attributes;
Terminal attributes (type IFT\$STORE_TERMINAL_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier and attribute value for each record.
The call cannot change the values of the following attributes:
- abort_line_char
 - backspace_char
 - cancel_line_char
 - network_control_char
 - output_flow-control
 - page_length
 - page_width
 - parity
 - pause_break_character
 - terminal_class
 - terminate_break_character
- status:** VAR of ost\$status;
Status variable.

Condition	ife\$cr_idle_range
Identifiers	ife\$current_job_not_interactive ife\$invalid_key_for_request ife\$lf_idle_range ife\$no_format_effectors_range ife\$no_transp_delim_selection ife\$page_wait_range ife\$prompt_file_id_not_found ife\$prompt_file_id_not_term ife\$prompt_file_name_ill_formed ife\$prompt_file_name_not_found ife\$prompt_file_name_not_term ife\$prompt_string_size_range ife\$special_editing_range ife\$transp_char_select_range ife\$transp_count_select_range ife\$transp_delim_count_range ife\$transp_timeout_select_range ife\$transparent_mode_range ife\$unknown_attribute_key ife\$unknown_input_device ife\$unknown_output_device
Remarks	The terminal attribute values specified by the call are used during the instance of open of the file. The values are discarded when the file is closed.

IFP\$FETCH_TERMINAL

Purpose	Returns terminal attribute values after the file is opened.
Format	IFP\$FETCH_TERMINAL (file_identifier, attributes, status)
Parameters	<p>file_identifier: amt\$file_identifier; File identifier returned when the file is opened.</p> <p>attributes: VAR of ift\$get_terminal_attributes; Terminal attributes (type IFT\$GET_TERMINAL_ATTRIBUTES). You must allocate a record in the adaptable array for each terminal attribute to be specified and specify an attribute identifier for each record; the procedure returns the attribute source and value in the record. The attribute sources are listed in table 5-1.</p> <p>status: VAR of ost\$status; Status variable.</p>
Condition Identifiers	<p>ife\$current_job_not_interactive</p> <p>ife\$unknown_attribute_key</p>
Remarks	The request returns the terminal attribute values that are currently in effect for this instance of open.

Terminal Attributes

As well as the set of file attributes described in chapter 6, Defining File Attributes, each local file name associated with the interactive terminal device class has a set of terminal attributes. The system uses terminal attribute values when processing interactive I/O.

The procedure calls described in this chapter specify terminal attribute values by allocating an array of records and assigning an attribute identifier to the key field of each record.

The `null_attribute` identifier is used to indicate that the record is to be ignored; it indicates that the call should not copy a value to the record or store a value from the record.

The following are descriptions of the terminal attributes. Further description of their functions can be found in the SCL System Interface manual. The default values set by NAM and NOS/VE are listed in tables 5-2 and 5-3.

abort_line_character

Character that, when entered as the only character on a line, discards the current output line.

Set by SET_TERMINAL_ATTRIBUTES command only.

backspace_character

Character that, when entered in an input line, discards the previous input character.

Set by SET_TERMINAL_ATTRIBUTES command only.

cancel_line_character

Character that discards the previous input line.

Set by SET_TERMINAL_ATTRIBUTES command only.

carriage_return_idle

Number of idle characters sent after a carriage return (0 through 99). The idle characters sent enable proper operation of mechanical printers.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

echoplex

Indicates whether each input character is automatically echoed back to the terminal.

TRUE

Selects echoplex.

FALSE

Deselects echoplex.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

eof_string

When this string is entered as a separate physical input line, the line serves the function of an end-of-information mark on the input file.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL_ATTRIBUTES, and IFP\$STORE_TERMINAL.

input_device

Input device.

IFC\$KEYBOARD_INPUT

Terminal keyboard input.

IFC\$PAPER_TAPE_INPUT

Paper tape reader input.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

line_feed_idle

Number of idle characters sent after a line feed (0 through 99). Like carriage_return_idle, the idle characters sent enable proper operation of mechanical printers.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

network_control_character

Character which, when entered as the first character on a line, causes the line to be processed as a network command rather than transmitted as data.

Set by SET_TERMINAL_ATTRIBUTES command only.

TERMINAL ATTRIBUTES

no_format_effectors

Indicates whether the system processes the first character of each output line as a character or a format effector.

TRUE

Do not process the first character as a format effector; pass the character to the terminal.

FALSE

Process the first character as a format effector.

Set by IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

NOTE

Setting the *no_format_effectors* to TRUE for the terminal file will not affect the *no_format_effectors* attribute for the prompt file. You must set this attribute separately for the prompt file to enforce or suppress format effectors.

null_attribute

Used to fill space in the attributes list. No attribute field generated; no value returned.

Set by IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

output_device

Output device used by terminal.

IFC\$DISPLAY_OUTPUT

Output displayed.

IFC\$PRINTER_OUTPUT

Output printed.

IFC\$PAPER_TAPE_OUTPUT

Output punched on paper tape.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

output_flow_control

Indicates whether the network is to allow the terminal to regulate the flow of output data to the terminal. When set to TRUE, the network suspends output data when an ASCII DC3 character is received from the terminal. Output resumes when the terminal sends an ASCII DC1 character. Neither the DC1 nor the DC3 character will be transmitted as data. Set by SET_TERMINAL_ATTRIBUTES command only.

page_length

Number of lines on display device (1 through 4,398,046,511,103). NAM interprets any value greater than 255 as meaning unlimited page length. Set by SET_TERMINAL_ATTRIBUTES and IFP\$TERMINAL.

page_wait

Indicates whether output is suspended at the end of a page and subsequently resumed when the user enters a carriage return.

TRUE

Waits at end of page.

FALSE

Does not wait at end of page.

The page length is specified by the page_length attribute. This attribute is effective only if the output_device attribute value is IFC\$DISPLAY_OUTPUT.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

page_width

Number of characters in a line on the display device (1 through 65,535). NAM interprets any value greater than 255 as meaning unlimited page width.

Set by SET_TERMINAL_ATTRIBUTES and IFP\$TERMINAL.

TERMINAL ATTRIBUTES

parity

Parity type.

IFC\$EVEN_PARITY

Performs even parity check.

IFC\$ODD_PARITY

Performs odd parity check.

IFC\$NO_PARITY

No parity check is performed; the parity bit is cleared if the `transparent_mode` attribute value is FALSE.

IFC\$ZERO_PARITY

No parity check is performed; the parity bit is always cleared.

Set by SET_TERMINAL_ATTRIBUTES command only.

pause_break_character

Character that, when entered as the only character on a line, causes a `pause_break` condition.

Set by SET_TERMINAL_ATTRIBUTES command only.

prompt_file

Local file name of the file to which the prompt string is written.

Unless the `prompt_file_id` attribute is set, the system opens the file specified by the `prompt_file` attribute when the task issues its first `get` call for the terminal file.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

prompt_file_id

File identifier of the open file to which the prompt string is written.

A task uses this attribute when it wants to open the prompt file with additional file attribute values specified on the AMP\$OPEN call. To do so, it does the following:

1. Opens the prompt file; AMP\$OPEN returns a file identifier.
2. Specifies the returned file identifier as the *prompt_file_id* terminal attribute.

If the task does not specify the file identifier as the *prompt_file_id*, the system is not aware that the task has already opened the prompt file; it performs its own open of the prompt file using the name provided by the *prompt_file* attribute.

Set by IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

NOTE

If the task specifies a *prompt_file_id* attribute value on an IFP\$TERMINAL or RMP\$REQUEST_TERMINAL call, the prompt file does not revert to the file specified by the *prompt_file* attribute until the task terminates. If the task specifies the *prompt_file_id* attribute on an IFP\$STORE_TERMINAL after the file is open, the prompt file reverts when the file is closed or when the task terminates.

prompt_string

Record describing the string output to the prompt file when a task issues a get call to the terminal file (type IFT\$PROMPT_STRING).

Field Content

size String length (0 through 31). (If the string length is zero, no prompt string is output.)

value String

If transparent mode is selected, no prompting is performed.

If one or more AMP\$PUT_PARTIAL calls have written a record that has not yet been terminated and sent to the terminal, a get call sends the partial record. If a prompt string is defined, it is appended to the partial record. The processing sequence is as follows:

1. The task issues one or more AMP\$PUT_PARTIAL calls with AMC\$START or AMC\$CONTINUE specified to write the first parts of an output record.
2. The task issues a get call. The system appends the prompt string to the partial output record, terminates the record, displays the prompt record at the terminal, and then waits for input.

If the no_format_effectors attribute is TRUE, the first character of the prompt string is interpreted as data, not as a format effector. If the no_format_effectors attribute is FALSE, the first character of the prompt string is interpreted as a format effector.

NOTE

If the first character of the prompt string is processed as a format effector, it is removed from the string before the string is appended to a pending partial output record and written to the prompt file.

special_editing

Determines whether the `cancel_line` character, `backspace_character`, and `line_feed_idle` character edit a line or are passed to the task as input data (boolean).

TRUE

Selects special editing.

FALSE

Deselects special editing.

Set by `IFP$TERMINAL`, `RMP$REQUEST_TERMINAL`, and `IFP$STORE_TERMINAL`.

terminal_class

Class of terminal used.

<code>IFC\$TTY_CLASS</code>	M3x teletypewriters.
<code>IFC\$C75x_CLASS</code>	CDC 75x or 713 terminals.
<code>IFC\$C721_CLASS</code>	CDC 721 terminals.
<code>IFC\$I1741_CLASS</code>	IBM 2741 terminals.
<code>IFC\$TTY40_CLASS</code>	M40 teletypewriters.
<code>IFC\$H2000_CLASS</code>	Hazeltine 2000 terminals.
<code>IFC\$X364_CLASS</code>	ANSII x3.64 terminals.
<code>IFC\$T4010_CLASS</code>	Tektronix 4010 terminals and CDC 721 and 722 terminals.
<code>IFC\$HASP_CLASS</code>	HASP protocol terminals.
<code>IFC\$C200UT_CLASS</code>	CDC 200 user terminals.
<code>IFC\$C711_CLASS</code>	CDC 711 terminals.
<code>IFC\$C714_CLASS</code>	CDC 714 terminals.
<code>IFC\$C73X_CLASS</code>	CDC 73x terminals.
<code>IFC\$I1780_CLASS</code>	IBM 2780 terminals.
<code>IFC\$I3780_CLASS</code>	IBM 3780 terminals.

Set by `SET_TERMINAL_ATTRIBUTES` command only.

terminal_name

Terminal name (7-character string).

Set by NOS/VE.

terminate_break_character

Character that, when entered as the only character on a line, causes a terminate_break condition.

Set by SET_TERMINAL_ATTRIBUTES command only.

transparent_delim_selection

Record specifying the conditions that end a transparent input block. If a field is TRUE, the condition is used; if it is FALSE, the condition is not used. At least one condition must be selected.

Field	Content
enable_end_delimiter	Transparent input delimiter (boolean). The delimiter is specified by the transparent_end_character attribute.
enable_end_count	Character count (boolean). The count is specified by the transparent_end_count attribute.
enable_time_out	Time period used (boolean).

The first of the selected conditions encountered ends the transparent input block. The condition does not end transparent input mode; transparent input mode ends when the transparent_mode attribute is set to FALSE.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

transparent_end_character

Character that delimits transparent input if selected by transparent_delim_selection attribute.

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

transparent_end_count

Character count that delimits transparent input if selected by transparent_delim_selection attribute (1 through 4,096).

Set by SET_TERMINAL_ATTRIBUTES, IFP\$TERMINAL, RMP\$REQUEST_TERMINAL, and IFP\$STORE_TERMINAL.

transparent_mode

I/O mode in which all terminal code conversion is bypassed.

TRUE

Selects transparent mode.

FALSE

Deselects transparent mode.

In transparent mode, the system does not assume that output is ASCII characters; it delivers output as 8-bit codes without any terminal-dependent conversions. Transparent input is passed to the program without any preprocessing or editing (such as backspacing or line cancelling).

The *transparent_delim_selection* specifies the input block delimiter conditions for transparent mode.

Set by *IFP\$TERMINAL*, *RMP\$REQUEST_TERMINAL*, and *IFP\$STORE_TERMINAL*.

Table 5-2. Default Attribute Values for Asynchronous Terminal Classes

Attribute	Terminal Classes			
	TTY	C75X	I2741	C721
abort_line_character	\$CHAR(24)	\$CHAR(24)	' ('	\$CHAR(24)
backspace_character	BS	BS	BS	BS
cancel_line_character	\$CHAR(8)	\$CHAR(8)	\$CHAR(8)	\$CHAR(8)
carriage_return_idle	\$CHAR(24)	\$CHAR(24)	' ('	\$CHAR(24)
echoplex	2	0	8	0
input_device	FALSE	FALSE	N/A	FALSE
line_feed_idle	Keyboard	Keyboard	Keyboard	Keyboard
network_control_character	1	0	1	0
no_format_effectors	ESC	ESC	ESC	ESC
output_device	\$CHAR(27)	\$CHAR(27)	' % '	\$CHAR(27)
output_flow_control	FALSE	FALSE	FALSE	FALSE
page_length	Printer	Display	Printer	Display
page_wait	FALSE	FALSE	N/A	FALSE
page_width	0	24	0	30
parity	FALSE	FALSE	FALSE	FALSE
prompt_file	72	80	132	80
prompt_string.size	Even	Even	Odd	Even
prompt_string.value	'OUTPUT'	'OUTPUT'	'OUTPUT'	'OUTPUT'
special_editing	3	3	3	3
transparent_delim_selection.enable_end_character	' ? '	' ? '	' ? '	' ? '
transparent_delim_selection.enable_end_count	FALSE	FALSE	FALSE	FALSE
transparent_delim_selection.enable_time_out	TRUE	TRUE	TRUE	TRUE
transparent_end_character	FALSE	FALSE	FALSE	FALSE
transparent_end_count	FALSE	FALSE	FALSE	FALSE
transparent_mode	FALSE	FALSE	FALSE	FALSE
N/A = Not applicable	CR	CR	CR	CR
	\$CHAR(13)	\$CHAR(13)	\$CHAR(13)	\$CHAR(13)
	2044	2044	N/A	2044
	FALSE	FALSE	FALSE	FALSE

(Continued)

Table 5-2. Default Attribute Values for Asynchronous Terminal Classes
(Continued)

Terminal Classes			
TTY40	H2000	x364	T4010
\$CHAR(24)	\$CHAR(24)	\$CHAR(24)	\$CHAR(24)
N/A	BS \$CHAR(8)	BS \$CHAR(8)	BS \$CHAR(8)
\$CHAR(24)	\$CHAR(24)	\$CHAR(24)	\$CHAR(24)
1	0	0	0
FALSE	FALSE	FALSE	FALSE
Keyboard	Keyboard	Keyboard	Keyboard
3	3	0	0
ESC \$CHAR(27)	ESC \$CHAR(27)	ESC \$CHAR(27)	ESC \$CHAR(27)
FALSE	FALSE	FALSE	FALSE
Display	Display	Display	Display
FALSE	FALSE	FALSE	FALSE
24	27	24	35
FALSE	FALSE	FALSE	FALSE
74	74	80	74
Even 'OUTPUT'	Even 'OUTPUT'	Even 'OUTPUT'	Even 'OUTPUT'
3	3	3	3
' ? '	' ? '	' ? '	' ? '
FALSE	FALSE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE
CR \$CHAR(13)	CR \$CHAR(13)	CR \$CHAR(13)	CR \$CHAR(13)
2044	2044	2044	2044
FALSE	FALSE	FALSE	FALSE

Table 5-3. Default Attribute Values for Synchronous Terminal Classes

Attribute	Terminal Classes		
	HASP	C200UT	C711
abort_line_character	N/A	N/A	N/A
backspace_character	N/A	N/A	N/A
cancel_line_character	'('	'('	'('
carriage_return_idle	N/A	N/A	N/A
echoplex	N/A	N/A	N/A
input_device	Keyboard	Keyboard	Keyboard
line_feed_idle	N/A	N/A	N/A
network_control_character	'%'	'%'	'%'
no_format_effectors	FALSE	FALSE	FALSE
output_device	Display	Display	Display
output_flow_control	N/A	N/A	N/A
page_length	0	13	16
page_wait	N/A	TRUE	TRUE
page_width	80	80	80
parity	N/A	Odd	Odd
prompt_file	'OUTPUT'	'OUTPUT'	'OUTPUT'
prompt_string_size	3	3	3
prompt_string_value	'?'	'?'	'?'
special_editing	FALSE	FALSE	FALSE
transparent_delim_selection.enable_end_character	TRUE	TRUE	TRUE
transparent_delim_selection.enable_end_count	FALSE	FALSE	FALSE
transparent_delim_selection.enable_time_out	FALSE	FALSE	FALSE
transparent_end_character	CR \$CHAR(13)	CR \$CHAR(13)	CR \$CHAR(13)
transparent_end_count	2044	2044	2044
transparent_mode	FALSE	FALSE	FALSE

(Continued)

Table 5-3. Default Attribute Values for Synchronous Terminal Classes
(Continued)

Terminal Classes			
C714	73x	I2780	I3780
N/A	N/A	N/A	N/A
N/A	N/A	N/A	N/A
'('	'('	N/A	N/A
N/A	N/A	N/A	N/A
N/A	N/A	N/A	N/A
Keyboard	Keyboard	N/A	N/A
N/A	N/A	N/A	N/A
'%'	'%'	'%'	'%'
FALSE	FALSE	FALSE	FALSE
Display	Display	N/A	N/A
N/A	N/A	N/A	N/A
16	13	0	0
TRUE	TRUE	N/A	N/A
80	80	80	120
Odd	Odd	N/A	N/A
'OUTPUT'	'OUTPUT'	'OUTPUT'	'OUTPUT'
3	3	3	3
'?'	'?'	'?'	'?'
FALSE	FALSE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	FALSE	FALSE
CR	CR	CR	CR
\$CHAR(13)	\$CHAR(13)	\$CHAR(13)	\$CHAR(13)
2044	2044	2044	2044
FALSE	FALSE	FALSE	FALSE

Special Considerations for Terminal File Processing

When using the file interface calls described in part II for a terminal file, you should be aware of the following special considerations.

File Attributes

Only the following file attributes are effective for terminal files. Differences in attribute processing for terminal files are listed.

- `access_mode`: Shorten and modify access have the same meaning as append access.
- `error_exit_name`, `file_access_procedure`, `return_option`, and `ring_` attributes: Same as for a local mass storage file.
- `file_organization`: Either sequential or byte_addressable file organization can be specified, but `byte_addressable` is processed the same as sequential.
- `internal_code`: Must be `AMC$ASCII` (the default attribute value).
- `page_length` and `page_width`: By default, the values for the respective terminal attributes are used as the corresponding file attributes. However, a file attribute definition command or call overrides the terminal attribute value.

File Access Information

An `AMP$FETCH_ACCESS_INFORMATION` call can return the following information for an open terminal file:

- `block_number`: The last `NAM` application block number accessed on the file.
- `last_op_status`: Always returns operation complete status.
- `error_status`, `file_position`, `last_access_operation`, and `previous_record_length`: Processed the same as for a local mass storage file.

The following access information items are not applicable to a terminal file: `current_byte_address`, `eoi_byte_address`, `previous_record_address`, `volume_number`, and `volume_position`.

File Interface Calls

The following file interface calls return an error status if a terminal file is specified on the call:

```
AMP$GET_SEGMENT_POINTER
AMP$SET_SEGMENT_EOI
AMP$SET_SEGMENT_POSITION
AMP$SKIP_TAPE_MARKS
AMP$WRITE_TAPE_MARK
```

The following file interface calls are ineffective (act as no-ops) if a terminal file is specified on the call:

```
AMP$SKIP
AMP$REWIND
AMP$WRITE_END_PARTITION
```

The following file interface calls are effective for terminal files. Special processing considerations are noted.

- **AMP\$RETURN:** Discards the file definition within the job. The connection of the job to the terminal and of other files to the terminal is not affected.
- **AMP\$OPEN:** The specified access level must be `AMC$RECORD_ACCESS`.
- **AMP\$CLOSE:** Flushes undelivered output to the terminal.
- **AMP\$GET_NEXT:** The length of the input line is returned in the `transfer_count` parameter variable.
- **AMP\$GET_PARTIAL:** The accumulated length of the input line is returned in the `record_length` parameter variable.
- **AMP\$GET_DIRECT** and **AMP\$PUT_DIRECT:** The `file_organization` attribute must be `AMC$BYTE_ADDRESSABLE`.
- **AMP\$PUT_NEXT** and **AMP\$PUT_PARTIAL:** Processed the same as for a mass storage file.
- **AMP\$FLUSH:** Delivers all output data to the terminal before returning control to the task.

Terminal Input

A terminal input line is terminated by a RETURN key. The RETURN is not passed to the task as part of the input data. Unless transparent mode is selected, the input data is edited (backspace and cancel_line characters are interpreted) before the input data is passed to the task.

To terminate interactive data to a task requesting terminal input, the string '*EOI' can be entered.

Line feeds within an input line are ignored. For example, suppose a user enters two lines, the first ending with a line feed and the second with a RETURN, as follows:

```
line 1 [LINE FEED]
line 2 [RETURN]
```

The input is passed to the task as a single line as follows:

```
line 1line 2
```

If transparent mode is not selected, the terminal codes are interpreted as character codes in the 128-character ASCII set. If transparent mode is selected, the terminal transfers data as 8-bit frames using whatever code the terminal sends on its communication line. Input blocks are terminated by the selected transparent input conditions, not by RETURNS.

NOS/VE allows more than one task in a job to be reading from the same terminal at the same time. However, NOS/VE performs no input request queueing for the tasks. When a task issues a get request, a full input line is delivered. A sequence of AMP\$GET_PARTIAL calls is satisfied from the same input line. File positioning information is separate for each task.

When a task has more than one concurrent instance of open of a terminal file, the file positioning information is shared for all instances of open. Therefore, AMP\$GET_PARTIAL calls for different instances of open could read parts of the same input line.

Typed Ahead Input

Terminal input can be typed ahead; that is, the user can enter input data before the task is ready to process it. The system queues the input lines until the task requests the input.

The number of lines that can be queued depends on the NAM terminal definition. NAM can queue up to 20 lines of input; NOS/VE can queue 5 additional lines. When you reach the type-ahead limit, NAM sends the following message:

```
WAIT..
```

You must then wait until the task accepts the input you have entered. You cannot interrupt the task (with a `pause_break` or `terminate_break` character) until the task accepts the input.

Terminal Output

Unless transparent mode is selected, output sent to a terminal file is assumed to be character codes of the 128-character ASCII set, and the system performs any needed code conversions for the terminal display. In transparent mode, the system sends the output data as a stream of 8-bit codes without any conversion.

NOS/VE allows more than one task in a job to be writing to the same terminal at the same time. However, NOS/VE performs no output request queuing for the tasks. The terminal usually receives output lines in the order that tasks create them.

When a task issues a put request, a full output line is delivered. A sequence of AMP\$PUT_PARTIAL calls is written as a single logical line unless the line exceeds 2,043 bytes. If a line exceeds 2,043 bytes, AMP\$PUT_PARTIAL calls from different tasks could write data as part of the same logical line. File positioning information is separate for each task.

When a task has more than one instance of open of a terminal file at the same time, the file positioning information is shared for all instances of open. Therefore, AMP\$PUT_PARTIAL calls for different instances of open could write parts of the same output line.

Format Effectors

Unless transparent mode is selected or the terminal attribute no_format_effectors is TRUE, the system processes the first character of an output line as a format effector. A format effector controls the vertical spacing of output at the terminal.

The following are the format effector constant identifiers, characters (in parentheses), and their effect:

- IFC\$PRE_PRINT_SPACE_1 (): Spaces down one line before printing.
- IFC\$PRE_PRINT_SPACE_2 (0): Spaces down two lines before printing.
- IFC\$PRE_PRINT_SPACE_3 (-): Spaces down three lines before printing.
- IFC\$PRE_PRINT_START_OF_LINE (+): Positions to the start of the current line before printing.
- IFC\$PRE_PRINT_HOME_CURSOR (*): Positions to the top of form before printing.
- IFC\$PRE_PRINT_HOME_CLEAR_SCREEN (1): Clears the screen before printing.
- IFC\$PRE_PRINT_NO_POSITIONING (,): Does nothing before printing.
- IFC\$POST_PRINT_SPACE_1 (,): Spaces down one line after printing.
- IFC\$POST_PRINT_START_OF_LINE (/): Positions to the start of the current line after printing.

The system converts the format effector character to the appropriate code for the terminal class.

Logical Lines

Each output record can contain more than one logical line. (Logical lines are not applicable in transparent mode.) Logical lines are separated by the ASCII US character. If the line contains only one logical line, do not include an ASCII US character at its end (because NOS/VE automatically adds an ASCII US to the end of the line). Each logical line begins with a format effector character (unless the attribute `no_format_effectors` is TRUE).

Page Wait

The page wait option causes the system to suspend output after sending a page of data. You then enter an empty line (by pressing the RETURN key only) to receive the next page of output. The empty line is not passed to the task as input. However, if you enter a nonempty line, the line is passed to the task as input; the entered line also restarts output.

The page wait option is effective when the `page_wait` terminal attribute is selected, the `output_device` attribute is `IFC$DISPLAY_OUTPUT` or `IFC$PRINTER_OUTPUT`, and a value has been defined for the `page_length` attribute unless in transparent mode. If the `page_length` attribute is undefined, the page length is assumed to be infinite and no page wait is performed. In transparent mode, an output block is considered to be a page, so a `page_length` value is not required.

The `page_length` attribute is both a terminal attribute and a file attribute. Therefore, a task can change the page length used for an instance of open using a file attribute definition call.

Line Folding

Line folding causes a line to continue on the next physical line when its length reaches the `page_width` value. The logical line remains the same regardless of the number of physical lines used to output or input the line.

Definition of a `page_width` terminal attribute value enables line folding. If the `output_device` attribute is `IFC$DISPLAY_OUTPUT`, the system assumes the terminal performs the line folding. However, if the `output_device` attribute is `IFC$PRINTER_OUTPUT`, the system performs the line folding.

The `page_width` attribute is both a terminal attribute and a file attribute. Therefore, a task can change the page width used for an instance of open using a file attribute definition call.

Terminal Conditions

Within NOS/VE, a condition is an occurrence that interrupts normal task processing. The chapter on condition processing in the *CYBIL System Interface manual* provides a complete description of condition processing.

Conditions that are especially pertinent to interactive processing include the following:

- Entry of the `pause_break` or `terminate_break` characters.
- Determination that the job is approaching a resource limit, such as a time limit.

The system processes these conditions by performing the following steps:

1. The system attempts to pass the condition to a condition handler selected by the task.
2. If the task has no condition handler for the condition, the system determines whether an `SCL WHEN` statement has specified processing for the condition. (For more information on the `WHEN` statement, see the *SCL System Interface manual*.)
3. If no `WHEN` statement is in effect for the condition, the system processes the condition itself, as follows:
 - `Pause_break`: Discards any input not yet read by a task (including any typed-ahead data) and suspends all user activity in the job. You can then enter `SCL` commands (such as a command to determine the job status). You can then resume or terminate the job.
 - `Terminate_break`: Discards any input not yet read by a task (including any typed-ahead data), terminates all user activity in the job, and discards all output not yet delivered to the terminal. It does not, however, terminate tasks suspended by a previous `pause_break`. You can then enter a command to continue processing.
 - `Resource limit`: Discards any input not yet read by a task, suspends all user activity in the job, and sends a message to the terminal. You can then respond to the message.



Defining New File Attributes	6-1
Defining Old File Attributes	6-3
Verifying Preserved Attribute Values	6-3
Defining Attributes for an Open File	6-3
Attribute Definition Calls	6-4
AMP\$FILE	6-5
AMPSTORE	6-8
Retrieving File Attributes	6-9
Attribute Specification	6-9
Attribute Sources	6-10
Returned Attributes	6-11
Retrieving Attributes for Connected Files	6-12
Retrieving File Characteristics	6-12
AMP\$GET_FILE_ATTRIBUTES	6-13
AMP\$FETCH	6-15
File Attribute Descriptions	6-16
List Attributes	6-42



Each file has the following characteristics:

- A local file name unique to a job.
- Assignment to a device class.
- A set of file attributes.

You can associate the name of a new file with a file attribute set, with a device class, or before it is opened.

A file attribute set is a set of values that the system references to determine how it processes a request to access the file. This chapter describes the specification of file attribute values.

The default device class assignment is mass storage. Part II of this manual describes device class assignment in detail.

Defining New File Attributes

A new file is a file that has never been opened. After a file has been opened, it becomes an old file.

A new mass storage file has no file space assigned to it. When a mass storage file is opened, it is assigned space, and the system stores certain file attributes with the file.

The file attributes stored with a mass storage file are called structural attributes or preserved attributes. These attributes determine the file structure and are preserved for the lifetime of the file. (You can change some preserved attribute values with a `CHANGE_FILE_ATTRIBUTES` command.)

The file attributes that are not stored with the file are called temporary attributes. The system discards temporary attribute values when the file is returned or closed.

The initial set of file attributes consists of default values which are system-defined. For a new file, you can change default attribute values and assign attribute values that do not have default values using the `SET_FILE_ATTRIBUTES` command and `AMP$FILE` and `AMP$OPEN` procedure calls.

You can specify more than one value for an attribute before a file is opened. An AMP\$FILE call discards all attribute values specified by previous AMP\$FILE calls; the resulting attribute set for the new file then consists of system_-defined default values and the values specified on the most recent AMP\$FILE call.

Values specified by SET_FILE_ATTRIBUTES commands override values set by AMP\$FILE calls. Values specified on the AMP\$OPEN call override values set by either SET_FILE_ATTRIBUTES or AMP\$FILE. Values specified by a SET_FILE_ATTRIBUTES or AMP\$OPEN call are cumulative; previously specified values are not discarded; the resulting attribute set consists of the previously specified values and the values specified on the call or command.

NOTE

When writing a program, you should consider whether the user of the program should be allowed to change a file attribute value using the SET_FILE_ATTRIBUTES command. To specify attribute values that the program user can change, use an AMP\$FILE call. To specify attribute values the program user cannot change, use the AMP\$OPEN call.

The value assigned to a structural attribute when you open the new file is the value preserved with the file.

Defining Old File Attributes

When you access an old mass storage file (a file that has previously been opened), its initial file attribute set consists of the preserved attribute values stored with the file, default values for temporary attributes defined by the system, and any values set by `SET_FILE_ATTRIBUTES` commands for the file. A task can also specify temporary attribute values that apply only to the current file access.

For an old file, if a `SET_FILE_ATTRIBUTES` command or an `AMP$FILE` request attempts to specify a value for a preserved attribute, the specified value will be ignored.

You can change the value of an attribute more than once before you open the old file. Values specified by `SET_FILE_ATTRIBUTES` commands override values set by `AMP$FILE` calls. Values specified on the `AMP$OPEN` call override values set by either `SET_FILE_ATTRIBUTES` or `AMP$FILE`.

Verifying Preserved Attribute Values

Besides changing temporary attribute values, the `AMP$OPEN` call also verifies preserved attribute values. If a preserved attribute value specified on the `AMP$OPEN` call does not match the actual preserved value, the procedure returns abnormal status (`AME$ATTRIBUTE_VALIDATION_ERROR`).

Defining Attributes for an Open File

As described earlier, the `SET_FILE_ATTRIBUTES` command and `AMP$FILE` call can specify attribute values before the file is opened. The `AMP$OPEN` call that opens the file can also specify attribute values.

After a file is opened, an `AMP$STORE` call can change attribute values. However, it is effective only for the `error_exit_procedure`, `error_limit`, and `message_control` attributes.

Attribute Definition Calls

Each call that defines file attributes specifies the file attribute values by specifying an array or a pointer to an array on the call. AMP\$FILE and AMP\$STORE calls specify the array; an AMP\$OPEN call specifies a pointer to an array.

To prepare the file attributes array, you first declare the array variable of the appropriate type. If you declare the parameter variable to be a pointer to the appropriate array type, you must also allocate the variable space with a PUSH or ALLOCATE statement.

Each file attributes array type is an adaptable array type. Therefore, you must fix the array size. The array should contain one element for each attribute to be specified.

For example, the following statements declare pointer variables for an AMP\$FILE call and an AMP\$OPEN call and then allocate space for the arrays:

```
VAR
    file_attributes_ptr: ^amt$file_attributes,
    open_attributes_ptr: amt$file_access_selections;

PUSH file_attributes_ptr: [1..1];
PUSH open_attributes_ptr: [1..1];
```

After declaring the variable type and allocating space for the variable, you initialize the tag field of each record to an attribute identifier and the value field to the attribute value.

An attribute identifier is the attribute name prefixed by AMC\$. The name of the attribute value field is the name of the attribute. (A listing of all attributes and the attributes valid for each call is provided later in this chapter.)

For example, the following statements initialize the file attributes variables for an AMP\$FILE call and an AMP\$OPEN call. The AMP\$FILE call specifies a value for the page_length attribute; the AMP\$OPEN call specifies a value for the page_width attribute. "Key" is the tag field.

```
file_attributes_ptr^[1].key := amc$page_length;
file_attributes_ptr^[1].page_length := 55;
open_attributes_ptr^[1].key := amc$page_width;
open_attributes_ptr^[1].page_width := 54;
```

The following are the procedure call descriptions for AMP\$FILE and AMP\$STORE. The AMP\$OPEN procedure call description is in chapter 7, Opening and Closing Files.

AMP\$FILE

Purpose Defines file attribute values for subsequent instances of open.

NOTE

You issue the AMP\$FILE call before you open the file. To change attributes of an open file, use an AMP\$STORE call.

An AMP\$FILE call discards any attribute values specified by previous AMP\$FILE calls specifying the file.

Format AMP\$FILE (local_file_name, file_attributes, status)

Parameters local_file_name: amt\$local_file_name;

Local file name.

file_attributes: amt\$file_attributes;

Array of attribute records. Each array record should contain an attribute identifier and an attribute value. The valid attributes for AMP\$FILE are listed in table 6-1.

status: VAR of ost\$status;

Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_attr_key
ame\$improper_file_attr_value
ame\$ring_validation_error

Remarks

- For a new file, an AMP\$FILE call can specify values for temporary attributes and preserved attributes. For an old file, an AMP\$FILE call can specify values only for temporary attributes.
- Calls to AMP\$FILE are not cumulative. If a task calls AMP\$FILE more than once before it opens the file, only the values specified on the last AMP\$FILE call are used.
- By specifying an attribute value with the AMP\$FILE procedure, you permit a user to override the value with a SET_FILE_ATTRIBUTES command; to prevent a task from overriding an attribute value, you must use an AMP\$OPEN call to specify the attribute value.
- The temporary attribute values specified on an AMP\$FILE call apply only to subsequent opens of the file within the issuing task until the file is returned. The values do not apply to previous or current instances of open of the file.

Table 6-1. Valid Attributes for Each File Attributes Call

Attribute	AMP\$ADD_ TO_FILE_		AMP\$GET_ FILE_		
	DESCRIPTION	AMP\$FETCH	AMP\$FILE	ATTRIBUTES	AMP\$OPEN AMP\$STORE
access_level		X			
access_mode		X	X	X	X
application_ info		X		X	
average_record_ length	X	X	X	X	X
block_type		X	X	X	X
character_ conversion	X	X	X	X	X
clear_space		X	X	X	X
collate_table	X	X			
collate_table_ name		X	X	X	X
compression_ procedure_name		X	X	X	X
data_padding	X	X	X	X	X
dynamic_home_ block_space		X	X	X	X
embedded_key	X	X	X	X	X
error_exit_name		X	X	X	X
error_exit_ procedure		X			X
error_limit		X	X	X	X
estimated_ record_count	X	X	X	X	X
file_access_ procedure		X	X	X	X
file_contents	X	X	X	X	X
file_length				X	
file_limit	X	X	X	X	X
file_organization		X	X	X	X
file_processor	X	X	X	X	X
file_structure	X	X	X	X	X
forced_write	X	X	X	X	X
global_access_ mode		X		X	
global_file_ address		X		X	
global_file_ name		X		X	
global_file_ position		X		X	

(Continued)

Table 6-1. Valid Attributes for Each File Attributes Call (Continued)

Attribute	AMP\$ADD_ TO_FILE DESCRIPTION	AMP\$FETCH	AMP\$FILE	AMP\$GET_ FILE ATTRIBUTES	AMP\$OPEN	AMP\$STORE
global_share_ mode		X		X		
hashing_ procedure_name		X	X	X	X	
index_levels	X	X	X	X	X	
index_padding	X	X	X	X	X	
initial_home_ block_count		X	X	X	X	
internal_code	X	X	X	X	X	
key_length	X	X	X	X	X	
key_position	X	X	X	X	X	
key_type	X	X	X	X	X	
label_type		X	X	X	X	
line_number	X	X	X	X	X	
loading_factor		X	X	X	X	
lock_expiration_ time		X	X	X	X	
logging_options		X	X	X	X	
log_residence		X	X	X	X	
max_block_ length	X	X	X	X	X	
max_record_ length	X	X	X	X	X	
message_ control		X	X	X	X	X
min_block_ length	X	X	X	X	X	
min_record_ length	X	X	X	X	X	
null_attribute	X	X	X	X	X	X
open_position		X	X	X	X	
padding_ character	X	X	X	X	X	
page_format	X	X	X	X	X	
page_length	X	X	X	X	X	
page_width	X	X	X	X	X	
permanent_file		X		X		
record_limit	X	X	X	X	X	
record_type	X	X	X	X	X	
records_per_ block	X	X	X	X	X	
return_option			X	X	X	
ring_attributes		X	X	X	X	
statement_ identifier	X	X	X	X	X	
user_info	X	X	X	X	X	
vertical_print_ density	X		X	X	X	

AMP\$STORE

Purpose Changes file attribute values for an open file.

NOTE

The AMP\$STORE procedure can only be called after the file is open. The attribute values specified on the call are applicable only to the instance of open specified on the call and are discarded when the file is closed.

Format AMP\$STORE (file_identifier, file_attributes, status)

Parameters **file_identifier:** amt\$file_identifier;
File access identifier returned by the AMP\$OPEN call that opened the file.

file_attributes: amt\$store_attributes;
Array of attribute records. Each array record should contain an attribute identifier and an attribute value. An AMP\$STORE call can only specify values for the error_exit_ procedure, error_limit, and message_control attributes.

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_attr_key
ame\$improper_file_attr_value
ame\$improper_file_id
ame\$ring_validation_error

For indexed sequential files only:
aae\$not_enough_permission

Remarks To retrieve attribute values specified by an AMP\$STORE call, use an AMP\$FETCH call. The AMP\$GET_FILE_ATTRIBUTES call does not return values set by an AMP\$STORE call.

Retrieving File Attributes

Besides specifying attribute values, you can also retrieve attribute values. Retrieving an attribute value allows you to change processing of the file according to the value returned.

Both `AMP$GET_FILE_ATTRIBUTES` and `AMP$FETCH` retrieve attribute values. The procedures have the following differences:

- You can call `AMP$GET_FILE_ATTRIBUTES` before or after you open the file; you can call `AMP$FETCH` only while the file is open.
- An `AMP$GET_FILE_ATTRIBUTES` call specifies the file by its local file name; the `AMP$FETCH` call specifies an instance of open of the file by the `file_identifier` returned by the `AMP$OPEN` call.
- To retrieve attribute values specified by an `AMP$STORE` call, you must use an `AMP$FETCH` call; `AMP$GET_FILE_ATTRIBUTES` does not return values stored by `AMP$STORE`.

Attribute Specification

Like file definition calls, each call to retrieve file attribute values has a `file_attributes` parameter. The `file_attributes` parameter must either name a static array or point to a dynamic array. In either case, the array must be of the type declared for the `file_attributes` parameter in the procedure declaration (`AMT$GET_ATTRIBUTES` for `AMP$GET_FILE_ATTRIBUTES` and `AMT$FETCH_ATTRIBUTES` for `AMP$FETCH`).

You declare and allocate the attributes array for an attribute retrieval call the same as for an attribute definition call except that you specify only the attribute identifier, not the attribute value. The procedure returns the attribute value in the value field of the record. (The only exception is the `collate_table` attribute; see the `collate_table` attribute description.)

For example, the following statements declare a pointer variable for an `AMP$GET_FILE_ATTRIBUTES` call, allocate space for an array containing two elements, and assign an attribute identifier to each record.

```
VAR
  get_attributes: ^amt$get_attributes;

PUSH get_attributes: [1..2];

get_attributes^[1].key := amc$page_length;
get_attributes^[2].key := amc$page_width;
```

After the AMP\$GET_FILE_ATTRIBUTES call is processed, an attribute value can be referenced as follows:

```
IF get_attributes^[2].page_width > 132 THEN
  line_folder;
IFEND;
```

If the attribute value is greater than 132, the IF statement calls a procedure named LINE_FOLDER.

Attribute Sources

Besides the attribute value, an attribute retrieval call also returns the attribute source. The attribute source indicates how the attribute value was defined.

The attribute retrieval call returns one of the identifiers listed in table 6-2 in the source field.

Table 6-2. File Attribute Sources

Identifier	Meaning
AMC\$UNDEFINED_ATTRIBUTE	The attribute does not have a default value, and no value has been specified for it.
AMC\$LOCAL_FILE_INFORMATION	The attribute value is determined by the job environment (returned attribute only).
AMC\$FILE_COMMAND	The attribute value was specified on a SET_FILE_ATTRIBUTES command.
AMC\$CHANGE_FILE_ATTRIBUTES	The attribute value was specified on a CHANGE_FILE_ATTRIBUTES command.
AMC\$FILE_REFERENCE	The attribute value was specified on the file reference. (For example, the open_position can be specified as \$BOI, \$EOI, or \$ASIS).
AMC\$FILE_REQUEST	The attribute value was specified on an AMP\$FILE call.
AMC\$ACCESS_METHOD_DEFAULT	The attribute value is the default value defined by the system.

(Continued)

Table 6-2. File Attribute Sources *(Continued)*

Identifier	Meaning
AMC\$OPEN_REQUEST	The attribute value was specified on an AMP\$OPEN call.
AMC\$ADD_TO_FILE_DESCRIPTION	The attribute value was specified on an AMP\$ADD_TO_FILE_DESCRIPTION call.
AMC\$STORE_REQUEST	The attribute value was specified on an AMP\$STORE call. (The AMP\$GET_FILE_ATTRIBUTES call cannot return this identifier.)

Returned Attributes

Certain file attributes cannot be specified by a user although the attribute retrieval calls can return the current value of these attributes. This manual refers to these attributes as returned attributes, rather than preserved or temporary attributes.

The system determines the values for returned attributes from the job environment, rather than a value specification. The attribute source identifier for a returned attribute is AMC\$LOCAL_FILE_INFORMATION.

The returned attributes include the following:

- application_info
- file_length
- global_access_mode
- global_file_address
- global_file_name
- global_file_position
- global_share_mode
- permanent_file

Retrieving Attributes for Connected Files

A `CREATE_FILE_CONNECTION` command or `CLP$CREATE_FILE_CONNECTION` call connects a subject file to a target file. If an attribute retrieval call specifies a subject file connected to one or more target files, the call returns the file attributes of the target file first connected to the subject file.

For example, suppose file `$ECHO` is the subject file first connected to the target file `MY_FILE` and then connected to the target file `YOUR_FILE`. An `AMP$GET_FILE_ATTRIBUTES` call that specifies `$ECHO` would return the attributes of `MY_FILE` because it is the first connected target file.

Retrieving File Characteristics

An `AMP$GET_FILE_ATTRIBUTES` call returns additional information besides file attribute values and sources. It also returns boolean values for the following parameters:

- `local_file`: Indicates whether the local file name is defined within the job.
- `old_file`: Indicates whether the file has previously been opened.
- `contains_data`: For a mass storage file, indicates whether the file contains data. (A file assigned to the terminal or null device classes always returns `FALSE`; a tape file always returns `TRUE`.)

If the call returns `FALSE` for both `local_file` and `old_file`, the file is not local to the job and has never been opened. In this case, the attribute values `AMP$GET_FILE_ATTRIBUTES` returns are the default attribute values; if the attribute does not have a default value, its value is `AMC$UNDEFINED_ATTRIBUTE`.

AMP\$GET_FILE_ATTRIBUTES

Purpose Returns file attribute values.

NOTE

The specified file can be open or closed when the AMP\$GET_FILE_ATTRIBUTES call is processed.

Format AMP\$GET_FILE_ATTRIBUTES (**local_file_name**, **file_attributes**, **local_file**, **old_file**, **contains_data**, **status**)

Parameters **local_file_name**: amt\$local_file_name;
Local file name.

file_attributes: VAR of amt\$get_attributes;

Array of attribute records. Each array record should contain an attribute identifier; the procedure AMP\$GET_FILE_ATTRIBUTES returns the attribute source and the attribute value in the record. The valid attributes are listed in table 6-1.

local_file: VAR of boolean;

Indicates whether the local file name is registered in the \$LOCAL catalog (boolean). TRUE is returned if the file is existent in the local catalog, FALSE if it is not.

old_file: VAR of boolean;

Indicates whether the file has been opened (boolean). TRUE is returned if the file has been opened, FALSE if it has not.

contains_data: VAR of boolean;

Indicates whether the file contains data (boolean). TRUE is returned if the file contains data, FALSE if it does not.

The call always returns FALSE if the file is assigned to a terminal or a null device. It always returns TRUE if the file is assigned to tape.

For indexed sequential files, contains_data is always TRUE after the file has been opened even if no data records have been written to the file. (Opening an indexed sequential file writes the internal file label.)

status: VAR of ost\$status;

Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

AMP\$GET_FILE_ATTRIBUTES

Condition ame\$improper_file_attr_key
Identifiers ame\$ring_validation_error

- Remarks**
- AMP\$GET_FILE_ATTRIBUTES does not return attribute values defined by AMP\$STORE calls.
 - If the AMP\$GET_FILE_ATTRIBUTES call specifies a subject file connected to one or more target files, the call returns the attributes of the target file to which the subject file was first connected.

AMP\$FETCH

Purpose Returns file attribute values.

NOTE

The instance of open specified by the file identifier on the AMP\$FETCH call must be open when the call is processed.

Format AMP\$FETCH (**file_identifier**, **file_attributes**, **status**)

Parameters **file_identifier**: amt\$file_identifier;
File identifier identifying the instance of open. AMP\$OPEN returns a file identifier when it opens a file.

file_attributes: VAR of amt\$fetch_attributes;
Array of attribute records. Each array record should contain an attribute identifier; AMP\$FETCH returns the attribute source and the attribute value in the record. The valid attributes are listed in table 6-1.

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_attr_key
ame\$improper_file_id
ame\$ring_validation_error

Remarks An AMP\$FETCH call returns attribute values specified by an AMP\$STORE call if the calls specify the same file identifier.

File Attribute Descriptions

Each of the following attribute descriptions provides the following information:

- Attribute name. (The name given is the name of the value field in the attribute record; the attribute identifier is the attribute name with the prefix AMC\$. For example, the attribute identifier for block_type is AMC\$BLOCK_TYPE.)
- Attribute purpose.
- Indicates whether the attribute is a preserved, temporary, or returned attribute.
- Valid attribute values.
- Default value for preserved and temporary attributes.

A preserved attribute is an attribute whose value is kept for the lifetime of the file. (You can change some preserved attribute values with a CHANGE_FILE_ATTRIBUTES command.) A temporary attribute is an attribute whose value is discarded after the file is returned. A returned attribute is an attribute whose value cannot be specified by an attribute definition command or call but can be returned by an attribute retrieval command or call. The file attribute descriptions follow.

access_level

Indicates the level of file data access used for this instance of open (returned attribute). The user defines the attribute value on the AMP\$OPEN call.

Value: One of the following identifiers (type AMT\$ACCESS_LEVEL):

AMC\$RECORD

Record access.

AMC\$SEGMENT

Segment access (valid only for mass storage files whose file_organization attribute is not AMC\$INDEXED_SEQUENTIAL).

access_mode

Set of access modes allowed within the instance of open (temporary attribute).

Value: Set of access mode identifiers (type PFT\$USAGE_SELECTIONS).

PFC\$READ

Read access.

PFC\$SHORTEN

Shorten access.

PFC\$APPEND

Append access (required to write to a new file).

PFC\$MODIFY

Modify access.

PFC\$EXECUTE

Execute access.

The set can contain only access modes included in the `global_access_mode` set (see the `global_access_mode` attribute description).

Default value: The set of access modes defined by the `global_access_mode` attribute excluding PFC\$EXECUTE.

The attribute cannot be changed while the file is opened.

For more information on access modes, see chapter 3, Mass Storage File Management.

application_info

Access control information used by an application program (returned attribute).

The application information string can be specified on a PFP\$PERMIT call. The Source Code Utility (SCU) uses the application information to determine whether a user has authority to perform certain operations.

Value: 31-character string (type PFT\$APPLICATION_INFO).

Default value: 31 spaces.

For more information on SCU use of this attribute, see the SCL Source Code Management manual.

average_record_length (indexed sequential files only)

Estimate of the average record length in bytes (preserved attribute). If specified, the system uses the attribute value to calculate the block size used; it uses the attribute value only when opening a new file.

For ANSI fixed-length (F) records, the *average_record_length* should be the same as the *maxi_record_length*.

For variable (V) and undefined (U) records, the *average_record_length* value depends on whether the majority of the records are the same length.

- If most records are a specific length, set the attribute value to that length.
- If the record lengths are well distributed within a range of lengths, set the attribute value to the median record length (half of the records are longer, half are shorter).

Value: integer from 1 through AMC\$MAXIMUM_RECORD (type AMT\$AVERAGE_RECORD_LENGTH).

Default value: None. If no value is set for the attribute, the system uses the arithmetic mean of the *max_record_length* and *min_record_length* values to calculate block size. However, the system does not set the *average_record_length* attribute to that value.

For more information, see chapter 10, Accessing Indexed Sequential Files.

block_type (sequential or byte addressable files only)

Indicates whether the user or the system determines file blocking (preserved attribute).

Value: One of the following identifiers (type AMT\$BLOCK_TYPE):

AMC\$SYSTEM_SPECIFIED

Access method determines block size.

AMC\$USER_SPECIFIED

User determines block size.

Default value: AMC\$SYSTEM_SPECIFIED.

For more information, see File Blocking in chapter 9, Accessing Sequential and Byte Addressable Files.

character_conversion (tape files only)

Indicates whether the tape file data requires conversion to 8-bit ASCII code (preserved attribute).

NOTE

Currently, NOS/VE does not perform tape file character conversion. However, the `character_conversion` attribute is available for use by a program that intends to perform its own character conversion.

Value: Boolean value.

TRUE

The system converts the character code.

FALSE

The system does not convert the character code.

Default value: FALSE.

collate_table

Collation table (returned attribute). This attribute is used to fetch the collation table assigned to a file.

NOTE

To fetch the collation table, you specify a pointer in the `COLLATE_TABLE` field of the attribute record for an `AMP$FETCH` call. `AMP$FETCH` copies the collation table to the variable to which the pointer points. If you do not specify a pointer, the system attempts to use an undefined pointer and returns an error.

Value: Pointer of type `^AMT$COLLATE_TABLE`. Type `AMT$COLLATE_TABLE` has the following declaration:

```
ARRAY [CHAR] OF AMT$COLLATION_VALUE
```

Type `AMT$COLLATION_VALUE` is the integer subrange 0 through 255.

To determine the collating weight the table assigns to a particular character code, you use the character as the index into the table; the value at that position is the collating weight of that character. For example, assume an `AMP$FETCH` call has fetched the collation table of a file and stored it in an array variable `COLLATION_TABLE`. The following statement assigns the collating weight of A to integer variable `A_WEIGHT`:

```
A_WEIGHT := COLLATION_TABLE['A'];
```

Assume the statement assigns the value 0 to `A_WEIGHT`. This means that the collation table assigns the collating weight 0 to character A.

Default value: None.

For more information, see chapter 10, Accessing Indexed Sequential Files.

collate_table_name

Collation table name (preserved attribute). This attribute is used to specify a collation table for a file.

The attribute value is used only when the file is first opened. When the file is opened, the named collation table is stored in the file label. The collation table for the file cannot be changed after the file has been opened.

Value: 31-character program name (PMT\$PROGRAM_NAME).

NOTE

All letters in the name must be specified as uppercase letters.

The name can be that of a system-defined collation table or a user-defined collation table. Collation table definition is described in appendix E, Collation Tables for Indexed Sequential Files.

The names of the system-defined collation tables follow. The collating sequence for each table is listed in appendix E.

OSV\$ASCII6_FOLDED

CYBER 170 FORTRAN 5 default collating sequence; lowercase letters mapped to uppercase letters.

OSV\$ASCII6_STRICT

CYBER 170 FORTRAN 5 default collating sequence.

OSV\$COBOL6_FOLDED

CYBER 170 COBOL 5 default collating sequence; lowercase letters mapped to uppercase letters.

OSV\$COBOL6_STRICT

CYBER 170 COBOL 5 default collating sequence.

OSV\$DISPLAY63_FOLDED

CYBER 170 63-character display code collating sequence; lowercase letters mapped to uppercase letters.

OSV\$DISPLAY63_STRICT

CYBER 170 63-character display code collating sequence.

OSV\$DISPLAY64_FOLDED

CYBER 170 64-character display code collating sequence; lowercase letters mapped to uppercase letters.

OSV\$DISPLAY64_STRICT

CYBER 170 64-character display code collating sequence.

OSV\$EBCDIC

Full EBCDIC collation sequence.

OSV\$EBCDIC6_FOLDED

EBCDIC 6-bit subset supported by CYBER 170 COBOL 5 and SORT 5; lowercase letters mapped to uppercase letters.

OSV\$EBCDIC6_STRICT

EBCDIC 6-bit subset supported by CYBER 170 COBOL 5 and SORT 5.

Default value: None. You must specify a value for the `collate_table_name` attribute if you specify `AMC$INDEXED_SEQUENTIAL` as the `file_organization` attribute value and `AMC$COLLATED_KEY` as the `key_type` attribute value.

For more information, see chapter 10, Accessing Indexed Sequential Files.

data_padding (indexed sequential files only)

Percentage of empty space the system is to leave in each data block when writing records at file creation time. The empty space allows for easy file expansion during later file processing operations (preserved attribute).

The attribute value is used only when an indexed sequential file is created.

Value: 0 through 99 (type `AMT$DATA_PADDING`).

Default value: 0 (no padding).

For more information, see chapter 10, Accessing Indexed Sequential Files.

embedded_key (indexed sequential files only)

Indicates whether the primary key is stored in the record (preserved attribute).

Value: Boolean value.

TRUE

Primary key is located in the record.

FALSE

Primary key is located separately from the record.

Default value: TRUE.

For more information, see chapter 10, Accessing Indexed Sequential Files.

error_exit_name

Name of an error processing procedure (temporary attribute).

The name must be that of a procedure with the XDCL attribute within the global library set of the job or defined within the task.

For the attribute to be effective, you must specify the *error_exit_name* value before the file is opened or on the AMP\$OPEN call. The error processing procedure is loaded when the file is opened. To change the procedure while the file is open, you must use the *error_exit_procedure* attribute.

Value: 1 through 31-character procedure name (type PMT\$PROGRAM_NAME). The named procedure must be of type AMT\$ERROR_EXIT_PROCEDURE; that is, it must have the following parameter list:

(file_identifier: amt\$file_identifier;
VAR status: ost\$status)

Default value: None. If no error exit name is specified, the system does not search for an error processing procedure.

For more information, see Error Exit Procedure in chapter 7, Opening and Closing Files.

error_exit_procedure

Address of the current error processing procedure (temporary attribute).

You use this attribute to change the effective error processing procedure while the file is open. To clear the effective error processing procedure, specify a nil pointer for the attribute.

Value: Pointer variable of type ^AMT\$ERROR_EXIT_PROCEDURE. A procedure of type AMT\$ERROR_EXIT_PROCEDURE has the following parameter list.

(file_identifier: amt\$file_identifier;
VAR status: ost\$status)

Default value: None. The system continues to use the error processing procedure specified by the *error_exit_name* attribute when the file was opened, if one was specified.

For more information, see Error Exit Procedure in chapter 7, Opening and Closing Files.

error_limit (indexed sequential files only)

Maximum number of trivial errors that can occur before the trivial errors cause a fatal error (temporary attribute).

Value: Integer (type AMT\$ERROR_LIMIT). 0 means no error limit.

Default value: 0 (no error limit).

For more information, see chapter 10, Accessing Indexed Sequential Files.

estimated_record_count (indexed sequential files only)

Estimated number of records the file will hold (preserved attribute). The system uses the attribute value to calculate block size; it only uses the value when it first opens a new file.

Value: Integer (type AMT\$ESTIMATED_RECORD_COUNT).

Default value: If a value is defined for the record_limit attribute, the record_limit value is the default estimated_record_count. If the record_limit attribute is undefined, the default value is 100,000.

For more information, see chapter 10, Accessing Indexed Sequential Files.

file_access_procedure

Name of the file access procedure (FAP) called when the file is accessed (preserved attribute).

Value: 1 through 31-character procedure name (type PMT\$PROGRAM_NAME). The name must be that of a procedure declared with the XDCL attribute within the global library set of the job or defined within the task. The procedure must be a FAP as described in appendix D, File Access Procedures.

Default value: If the attribute does not have a value when the file is first opened, the file has no FAP associated with it. However, a CHANGE_FILE_ATTRIBUTE command can specify a FAP for the file.

file_contents

String describing the file contents (preserved attribute).

Value: The following string identifiers are defined by the system (type AMT\$FILE_CONTENTS):

AMC\$UNKNOWN_CONTENTS	'UNKNOWN'
AMC\$LIST	'LIST'
AMC\$LEGIBLE	'LEGIBLE'
AMC\$SOURCE	'SOURCE'

Default value: AMC\$UNKNOWN_CONTENTS.

file_length

Length of a mass storage file in bytes (returned attribute).

Value: Integer (type AMT\$FILE_LENGTH).

file_limit

Maximum file length in bytes (preserved attribute).

For files opened for record access, the end-of-information (EOI) must not exceed the file_limit value. If it does, the procedure returns abnormal status.

For files opened for segment access using a sequence or heap structure, the file_limit value is the maximum size of the sequence or heap. A page reference beyond file_limit causes a segment access condition.

Value: Integer (type AMT\$FILE_LIMIT).

Default value: 100,000,000 (the effective file byte limit).

file_organization

File organization (preserved attribute).

Value: One of the following identifiers (type AMT\$FILE_ORGANIZATION):

- AMC\$SEQUENTIAL
Sequential organization.
- AMC\$BYTE_ADDRESSABLE
Byte addressable organization.
- AMC\$INDEXED_SEQUENTIAL
Indexed sequential organization.

Default value: AMC\$SEQUENTIAL.

file_processor

String identifying the intended processor of the file (preserved attribute). It is used with the `file_contents` and `file_structure` values to identify the file contents.

The `file_processor` identifies the intended processor of the file data, not the file creator. For example, `AMC$CYBIL` indicates that the file contains input (source code) for the CYBIL compiler.

Value: The following string identifiers are defined by the system (type `AMT$FILE_PROCESSOR`).

<code>AMC\$UNKNOWN_PROCESSOR</code>	<code>'UNKNOWN'</code>
<code>AMC\$COBOL</code>	<code>'COBOL'</code>
<code>AMC\$CYBIL</code>	<code>'CYBIL'</code>
<code>AMC\$DEBUGGER</code>	<code>'DEBUGGER'</code>
<code>AMC\$FORTRAN</code>	<code>'FORTRAN'</code>
<code>AMC\$SCU</code>	<code>'SCU'</code>
<code>AMC\$CPU_ASSEMBLER</code>	<code>'CPU_ASSEMBLER'</code>
<code>AMC\$PPU_ASSEMBLER</code>	<code>'PPU_ASSEMBLER'</code>

Default value: `AMC$UNKNOWN_PROCESSOR`.

file_structure

String identifying the file structure (preserved attribute). It is used with the `file_contents` and `file_processor` values to identify the file contents.

Value: The following string identifiers are defined by the system (type `AMT$FILE_STRUCTURE`):

<code>AMC\$UNKNOWN_STRUCTURE</code>	<code>'UNKNOWN'</code>
<code>AMC\$DATA</code>	<code>'DATA'</code>
<code>AMC\$LIBRARY</code>	<code>'LIBRARY'</code>

Default value: `AMC$UNKNOWN_STRUCTURE`.

forced_write (indexed sequential files only)

Indicates whether the system copies modified blocks to mass storage immediately after modification or allows modified blocks to remain in memory until the next flush or close request (preserved attribute).

Value: One of the following identifiers (type AMT\$FORCED_WRITE):

AMC\$FORCED

The system writes each modified block to mass storage immediately after the block is modified.

AMC\$FORCED_IF_STRUCTURE_CHANGE

The system writes modified blocks to mass storage immediately after any structure change to the file that affects more than one block.

AMC\$UNFORCED

The system determines when to write modified blocks to mass storage. Modified blocks can remain in memory without a backup copy on mass storage.

Default value: AMC\$FORCED_IF_STRUCTURE_CHANGE.

global_access_mode

Indicates the set of valid access modes for the file (returned attribute). For an existing permanent file, the set of access modes is determined when the file is attached. For a temporary file or a new permanent file, the set includes all usage modes.

Value: Set of any (including none) of the following constant identifiers (type PFT\$USAGE_SELECTIONS):

PFC\$READ

Read access.

PFC\$SHORTEN

Shorten access.

PFC\$APPEND

Append access (required to write to a new file).

PFC\$MODIFY

Modify access.

PFC\$EXECUTE

Execute access.

Default value: For permanent files, the set of access modes specified when the file is attached. For temporary files, the set containing all access modes (read, modify, append, shorten, and execute).

global_file_address

File byte address attained by the last get, put, AMP\$SET_SEGMENT_EOI, or AMP\$SET_SEGMENT_POSITION call to the file (returned attribute).

Value: Integer (type AMT\$FILE_BYTE_ADDRESS).

For more information, see Sharing a Segment Access File in chapter 8, Accessing a File as a Memory Segment.

global_file_name

File name uniquely identifying the file (returned attribute). The system generates the name for the file when it creates the file. The global file name allows a program to determine whether files having different local file names are actually the same file.

Value: Packed record having the following fields (type OST\$BINARY_UNIQUE_NAME):

processor_ serial_number	Integer (type PMT\$CPU_SERIAL_NUMBER)
processor_ model_number	One of the following constant identifiers (type PMT\$CPU_MODEL_NUMBER):
	PMC\$CPU_MODEL_P1
	PMC\$CPU_MODEL_P2
	PMC\$CPU_MODEL_P3
	PMC\$CPU_MODEL_P4
year	Integer from 1980 through 2047.
month	Integer from 1 through 12.
hour	Integer from 0 through 23.
day	Integer from 1 through 31.
minute	Integer from 0 through 59.
second	Integer from 0 through 59.
sequence_number	Integer from 0 through 9,999,999.

global_file_position

File position at completion of the last access request for the file (returned attribute). For more information, see Sharing a Segment Access File in chapter 8, Accessing a File as a Memory Segment.

Value: One of the following identifiers (type AMT\$GLOBAL_FILE_POSITION):

AMC\$BOI

Beginning-of-file.

AMC\$BOP

Beginning-of-partition.

AMC\$MID_RECORD

Within a record.

AMC\$EOR

End-of-record.

AMC\$EOP

End-of-partition.

AMC\$EOI

End-of-file.

global_share_mode

Indicates the valid share modes for the file (returned attribute). For a permanent file, the share modes are specified when the file is attached. Temporary files cannot be shared.

Value: Set of any number (including none) of the following constant identifiers (type PFT\$SHARE_SELECTIONS):

PFC\$READ

Read access.

PFC\$SHORTEN

Shorten access.

PFC\$APPEND


Append access.

PFC\$MODIFY


Modify access.

PFC\$EXECUTE

Execute access.

 *index_levels* (indexed sequential files only)

Target number of index levels (preserved attribute). The system uses the attribute value to calculate block size. The *index_levels* value is used only when the file is created.



Value: 1 through 15 (type AMT\$INDEX_LEVELS).

Default value: 2.

For more information, see chapter 10, Accessing Indexed Sequential Files.




index_padding (indexed sequential files only)

Percentage of index block space to be left empty when the file is created. The empty space allows easy file expansion (preserved attribute).

Value: 0 through 99 (type AMT\$INDEX_PADDING).

Default value: 0 (no padding).

For more information, see chapter 10, Accessing Indexed Sequential Files.

internal_code (sequential or byte addressable files only)

Character code of file (preserved attribute).

NOTE

Currently, NOS/VE does not perform character conversion. However, the *internal_code* attribute is available for use by a program that intends to perform its own character conversion.

Value: One of the following identifiers (type `AMT$INTERNAL_CODE`):

`AMC$AS6`

CYBER 170 6/12 ASCII code.

`AMC$AS8`

CYBER 170 8/12 ASCII code.

`AMC$ASCII`

8-bit ASCII code.

`AMC$BCD`

Binary coded decimal code.

`AMC$D64`

CYBER 170 64-character display code.

`AMC$EBCDIC`

9-bit EBCDIC tape code.

Default value: `AMC$ASCII`.

key_length (indexed sequential files only)

Primary key length in bytes (preserved attribute).

Value: Integer (type `AMT$KEY_LENGTH`). (For files with embedded keys, the value cannot be greater than the `minimum_record_length` value.)

Default value: No default value. When opening a new indexed sequential file, `AMP$OPEN` returns a fatal error if the attribute value is not set.

For more information, see chapter 10, Accessing Indexed Sequential Files.

key_position (indexed sequential files only)

Byte offset in the record where the primary key begins (preserved attribute). This attribute is ignored for files with nonembedded keys.

The value of `key_position + 1` defines the first byte of the primary key. For example, if `key_position` is set to three the primary key begins in the fourth byte of the record.

Value: 0 through `MAX_RECORD_LENGTH` (type `AMT$KEY_POSITION`). The sum of the `key_position` and `key_length` values cannot be greater than the `max_record_length` value.

Default value: 0 (beginning of record).

For more information, see chapter 10, Accessing Indexed Sequential Files.

key_type (indexed sequential files only)

Primary key type (preserved attribute).

Value: One of the following identifiers (type `AMT$KEY_TYPE`):

AMC\$UNCOLLATED_KEY

Keys (1 through 255 bytes) ordered byte-by-byte according to the ASCII character set sequence (listed in appendix B). The key can be a positive integer or a string of ASCII character codes.

AMC\$INTEGER_KEY

Integer keys (1 through 8 bytes) ordered numerically. The integer can be positive or negative.

AMC\$COLLATED_KEY

Collated character keys (1 through 255 characters) ordered using the collation table specified by the `collate_table_name` attribute. If `AMC$COLLATED_KEY` is specified, the `collate_table_name` attribute must also be specified.

Default value: `AMC$UNCOLLATED_KEY`.

For more information, see chapter 10, Accessing Indexed Sequential Files.

label_type (sequential or byte addressable files only)

Tape labels used (preserved attribute).

Value: Currently, the following constant identifier (type `AMT$LABEL_TYPE`):

AMC\$UNLABELLED

No labels.

Default value: `AMC$UNLABELLED`.

last_operation

Code indicating the latest operation the system has performed for the file (returned attribute).

Value of type `AMT$LAST_OPERATION`. The following lists file interface calls and the corresponding constant identifier declarations:

`AMP$ABANDON_KEY_DEFINITIONS``amc$abandon_key_definitions``AMP$ACCESS_METHOD``amc$access_method_req``AMP$APPLY_KEY_DEFINITIONS``amc$apply_key_definitions``AMP$ADD_TO_FILE_DESCRIPTION``amc$add_to_file_description_req``AMP$CLOSE``amc$close_req``AMP$COPY_FILE``amc$copy_file_req``AMP$CREATE_KEY_DEFINITION``amc$create_key_definition``AMP$DELETE_KEY``amc$delete_key_req``AMP$DELETE_KEY_DEFINITION``amc$delete_key_definition``AMP$FETCH``amc$fetch_req``AMP$FETCH_ACCESS_INFORMATION``amc$fetch_access_information_rq``AMP$FETCH_FAP_POINTER``amc$fetch_fap_pointer_req``AMP$FILE``amc$file_req``AMP$FLUSH``amc$flush_req`

AMP\$GET_DIRECT
amc\$get_direct_req

AMP\$GET_FILE_ATTRIBUTES
amc\$get_file_attributes_req

AMP\$GET_KEY
amc\$get_key_req

AMP\$GET_NEXT
amc\$get_next_req

AMP\$GET_NEXT_KEY
amc\$get_next_key_req

AMP\$GET_NEXT_PRIMARY_KEY_LIST
amc\$get_next_primary_key_list

AMP\$GET_PARTIAL
amc\$get_partial_req

AMP\$GET_PRIMARY_KEY_COUNT
amc\$get_primary_key_count

AMP\$GET_SEGMENT_POINTER
amc\$get_segment_pointer_req

AMP\$OPEN
amc\$open_req

AMP\$PUT_DIRECT
amc\$put_direct_req

AMP\$PUT_KEY
amc\$put_key_req

AMP\$PUT_NEXT
amc\$put_next_req

AMP\$PUT_PARTIAL
amc\$put_partial_req

AMP\$PUTREP
amc\$putrep_req

AMP\$REPLACE_KEY
amc\$replace_key_req

AMP\$RETURN

amc\$return_req

AMP\$REWIND

amc\$rewind_req

AMP\$SEEK_DIRECT

amc\$seek_direct_req

AMP\$SELECT_KEY

amc\$select_key

AMP\$SET_FILE_INSTANCE_ABNORMAL

amc\$set_file_inst_abnormal_req

AMP\$SET_LOCAL_NAME_ABNORMAL

amc\$set_local_name_abnormal_req

AMP\$SET_SEGMENT_EOI

amc\$set_segment_eoi_req

AMP\$SET_SEGMENT_POSITION

amc\$set_segment_position_req

AMP\$SKIP

amc\$skip_req

AMP\$SKIP_TAPE_MARKS

amc\$skip_tape_marks_req

AMP\$START

amc\$start_req

AMP\$STORE

amc\$store_req

AMP\$STORE_FAP_POINTER

amc\$store_fap_pointer_req

AMP\$WRITE_END_PARTITION

amc\$write_end_partition_req

AMP\$WRITE_TAPE_MARK

amc\$write_tape_mark_req

line_number (sequential or byte addressable files only)

Line number length and its location in the record (preserved attribute).

Leading and trailing blanks are acceptable, but blanks cannot be embedded in a line number. Line numbers must be in ascending order within a compilation unit.

Value: Record containing the following fields (type AMT\$LINE_NUMBER):

length

Number of bytes in the line number (integer from 1 through 6).

location

Byte within the line at which the line number begins (integer from 1 through 65,536).

Default value: None.

max_block_length

Maximum length of a file block in bytes (preserved attribute). The attribute is effective only with user-specified blocking.

Value: Integer from 1 through AMC\$MAXIMUM_BLOCK - 1 (type AMT\$MAX_BLOCK_LENGTH). For indexed sequential files, the range is 1 through 65,536; the system rounds up to the next power of 2 from 2,048 to 65,536, inclusive.

Default value: For sequential and byte addressable files, 4,128. For indexed sequential files, the system calculates an appropriate default value.

max_record_length

Maximum length of a file record in bytes (preserved attribute). The system only uses this attribute for indexed sequential files and for files with ANSI fixed length (F) records although certain products (such as Sort/Merge) use the attribute when processing other record types.

Value: Integer from 0 through AMC\$MAXIMUM_RECORD (type AMT\$MAX_RECORD_LENGTH). For indexed sequential files, the range is 1 through 65,497.

Default value: For sequential and byte addressable files, 256. For indexed sequential files, no default value is provided; AMP\$OPEN returns a fatal error if an attribute value is not specified when the file is created.

message_control (indexed sequential files only)

Indicates that additional information is written on the \$ERRORS file (temporary attribute).

Value: Set of one or more of the following identifiers indicating the information written (type AMT\$MESSAGE_CONTROL):

AMC\$TRIVIAL_ERRORS

Trivial errors logged (errors of severity ERROR).

AMC\$MESSAGES

Informative messages logged.

AMC\$STATISTICS

Statistics logged.

Default value: Null set (only fatal error messages are logged).

For more information, see chapter 10, Accessing Indexed Sequential Files.

min_block_length (sequential or byte addressable files only)

Minimum length of a file block in bytes (preserved attribute). The attribute is effective only with user-specified blocking.

Value: Integer from 18 through AMC\$MAXIMUM_BLOCK-1 (type AMT\$MIN_BLOCK_LENGTH). Seventeen bytes is the longest tape noise block size.

Default value: 18.

min_record_length (indexed sequential files only)

Minimum record length in bytes (preserved attribute).

Value: Integer from 0 to AMC\$MAXIMUM_RECORD (type AMT\$MIN_RECORD_LENGTH). For indexed sequential files, the value must be in the range 0 though 65,497, but not greater than the max_record_length value.

Default value: For ANSI fixed-length (F) records, the default value is the max_record_length value. For indexed sequential files using embedded keys, the default value is the sum of the key_position and key_length values. Otherwise, the default value is 1.

For more information, see chapter 10, Accessing Indexed Sequential Files.

null_attribute

Attribute identifier (AMC\$NULL_ATTRIBUTE) that indicates that the content of the attribute record is to be ignored.

open_position

Positioning required when the system opens the file (temporary attribute).

Value: One of the following identifiers (type AMT\$OPEN_POSITION):

AMC\$OPEN_NO_POSITIONING

File opened at current position (ASIS). This value opens an indexed sequential file at its beginning-of-information (BOI).

AMC\$OPEN_AT_BOI

File opened at its beginning-of-information.

AMC\$OPEN_AT_EOI

File opened at its end-of-information.

If the file is an old file and the only valid access mode to the file is append, AMC\$OPEN_AT_EOI is the only valid open position.

Default value: For all files other than file OUTPUT, AMC\$OPEN_AT_BOI. For file OUTPUT, AMC\$OPEN_AT_EOI.

The *open_position* specified on a file reference overrides all specifications of that attribute except an *open_position* value specified on an AMP\$OPEN call. For example, if a file is referenced as \$USER.MY_FILE.\$BOI, it is opened at its beginning-of-information unless the AMP\$OPEN call specifies another *open_position*. For more information on file references, see the SCL Language Definition manual.

padding_character (sequential or byte addressable files only)

Character used to pad a short fixed-length (F) record (preserved attribute).

Value: An ASCII character (type AMT\$PADDING_CHARACTER).

Default value: Space.

page_format (for listing file use)

Determines the listing format (preserved attribute). By system convention, a processor specifies this value for its listing file and then calls internal system routines that use the value to format the listing. A user can display this attribute value to determine how a listing is formatted before sending it to a printer.

The meanings listed for the attribute values describe how the internal system routines interpret the values.

Value: One of the following identifiers (type `AMT$PAGE_FORMAT`):

`AMC$CONTINUOUS_FORM`

No page numbering; title inserted at the beginning of each new type of information (source, errors, and so forth) and at the beginning of each page. (The `page_length` is determined by the `page_length` attribute value.) This is the recommended value for files to be listed at a terminal.

`AMC$BURSTABLE_FORM`

Pages are numbered; top-of-form character and title inserted at the beginning of each new type of information (source, errors, and so forth) and at the beginning of each page. This is the commended value for files to be listed on a forms printer with a page eject required for each page.

`AMC$NON_BURSTABLE_FORM`

Pages are numbered; title inserted at the beginning of each new type of information (source, errors, and so forth) and at the beginning of each page. Insertion of a top-of-form character before the title depends on the amount of space left on the page. If sufficient space remains to trip-space and print the title and three lines of data, the top-of-form character is omitted. This value shortens listing printed on a forms printer; each page is filled before a page eject is performed.

Default value: For terminal files, `AMC$CONTINUOUS_FORM`; for all other files, `AMC$BURSTABLE_FORM`.

page_length (sequential or byte addressable files only)

Number of lines on a page (preserved attribute).

Value: Integer from 1 through `AMC$FILE_BYTE_LIMIT` (type `AMT$PAGE_LENGTH`).

Default value: For terminal files, the maximum file length (`AMC$FILE_BYTE_LIMIT`). For all other files, the `vertical_print_density` value multiplied by ten. (The default value assumes a 10-inch print form.)

page_width (sequential or byte addressable files only)

Number of characters on a line (preserved attribute).

Value: Integer value from 1 through `AMC$MAX_PAGE_WIDTH` (type `AMT$PAGE_WIDTH`).

Default value: For a print line, 132; for a terminal line, the width of the terminal screen.

permanent_file

Indicates whether the file is permanent or temporary (returned attribute).

Value: Boolean value.

TRUE

File is permanent.

FALSE

File is temporary.

record_limit (indexed sequential files only)

Maximum number of records in the file (preserved attribute).

Value: Integer from 1 through AMC\$FILE_BYTE_LIMIT (2⁴²-1) (type AMT\$RECORD_LIMIT).

Default value: AMC\$FILE_BYTE_LIMIT (2⁴²-1).

For more information, see chapter 10, Accessing Indexed Sequential Files.

record_type

Record type of file (preserved attribute).

Value: One of the following identifiers (type AMT\$RECORD_TYPE):

AMC\$VARIABLE

CDC variable-length (V) records.

AMC\$UNDEFINED

Undefined (U) records.

AMC\$ANSI_FIXED

ANSI fixed-length (F) records.

For indexed sequential files, V and U records are internally equivalent.

Default value: For sequential and byte addressable files, AMC\$VARIABLE; for indexed sequential files, AMC\$UNDEFINED; for files created with segment access, AMC\$UNDEFINED.

For more information on record types, see Record Types in chapter 9.

records_per_block (indexed sequential files only)

Estimated number of records each data block should contain (preserved attribute). The system uses the attribute value to calculate block size; it uses the value only when opening a new file. It does not use the value as a limit to the number of records that a block can contain.

Value: Integer from 1 to `AMC$MAX_RECORDS_PER_BLOCK` (type `AMT$RECORDS_PER_BLOCK`).

Default value: 2.

For more information, see chapter 10, Accessing Indexed Sequential Files.

return_option

Indicates when the file is implicitly detached (returned) to the system (temporary attribute). (You can explicitly detach a file with a `DETACH_FILE` command or an `AMP$RETURN` call.)

Value: One of the following identifiers (type `AMT$RETURN_OPTION`):

`AMC$RETURN_AT_CLOSE`

Detach when the task closes the file and the job does not have another instance of open for the file.

`AMC$RETURN_AT_JOB_EXIT`

Return when the job terminates.

NOTE

If the file cannot be detached when it is closed and the `return_option` `AMC$RETURN_AT_CLOSE` was specified, the task does not receive notification that the file is not detached.

Default value: `AMC$RETURN_AT_JOB_EXIT`.

ring_attributes

Three ring numbers (r1, r2, and r3) defining the ring brackets of the file (preserved attribute).

- Write bracket: 1 through r1.
- Read bracket: 1 through r2.
- Execute bracket: r1 through r2.
- Call bracket: r2+1 through r3.

The ring numbers cannot be lower than the ring number of the caller that opens the file. If a new file is created by a file reference, its *ring_attributes* are those of the provider of the file reference specification.

Value: Record with three integer fields r1, r2, and r3 (type AMT\$RING_ATTRIBUTES).

Default value: All three ring numbers are the ring number of the AMP\$OPEN caller. If the file has not yet been opened, the attribute value is undefined.

statement_identifier (sequential or byte addressable files only)

Statement identifier length and its location in the line (preserved attribute).

Value: Record containing the following fields (type AMT\$STATEMENT_IDENTIFIER):

length

Number of characters in the statement identifier (integer from 1 through 17).

location

Character position of the first digit of the statement identifier (integer from 1 through AMC\$MAX_PAGE_WIDTH).

Default value: None.

user_info

String that the system maintains as a file attribute (preserved attribute). This attribute is used for interstate communication (see the CYBIL System Interface manual).

Value: 32-character string (type AMT\$USER_INFO).

Default value: 32 blanks.

vertical_print_density (sequential or byte addressable files only)

Number of lines printed per inch (preserved attribute). A program can reference this attribute value to determine the appropriate print density and then add format effectors to select and deselect the print density.

The NOS/VE product set does not add format effectors to control the print density.

Value: Integer from 6 through 12 (AMT\$VERTICAL_PRINT_DENSITY).

Default value: 6.

List Attributes

When creating a file to be printed, the task should set the following attributes:

- *file_contents* (must be AMC\$LIST or AMC\$UNKNOWN_CONTENTS).
- *file_structure* (must be AMC\$DATA or AMC\$UNKNOWN_STRUCTURE).
- *page_format*.
- *page_length*.
- *page_width*.
- *vertical_print_density*.

The NOS/VE product set does not add format effectors to control the print density. Currently, to change the print density from the default, the program must add format effectors to select and deselect a print density.

- File Identifiers 7-1
 - AMP\$OPEN 7-2
 - AMP\$CLOSE 7-5
- Access Validation 7-6
 - Ring Number Validation 7-6
 - Access Mode Validation 7-6
 - Open Position for Appending 7-6
 - Implicit Release of File Data 7-6
- Error Exit Procedure 7-7
 - Error Exit Procedure Attributes 7-7
- File Sharing 7-8
 - Reading a Shared File 7-8
 - Writing a Shared File 7-9
 - Retrieving Access Information 7-9
 - AMP\$FETCH_ACCESS_INFORMATION 7-16



Opening a file enables access to its data; closing a file prevents access to its data until the file is reopened. An AMP\$OPEN call opens a file; an AMP\$CLOSE call closes an opened file.

An instance of open corresponds to an AMP\$OPEN call for the file. Separate access information is maintained for each instance of open. The access information for an instance of open can be retrieved by an AMP\$FETCH_ACCESS_INFORMATION call.

For each instance of open, file data access is either through record access or segment access. The access level is specified on the AMP\$OPEN call.

File Identifiers

For each instance of open, AMP\$OPEN assigns a file identifier. File interface calls, then uses the file identifier to reference an instance of open. Separate file positioning information is kept for each file identifier.

An AMP\$CLOSE call closes only one instance of open, the instance of open specified by the file identifier on the call.

AMP\$OPEN

Purpose Prepares a local file for I/O.

NOTE

If the AMP\$OPEN call specifies a file with indexed sequential file organization, you must specify \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE as an alternate base library when expanding the source program decks.

Format AMP\$OPEN (local_file_name, access_level, access_selections, file_identifier, status)

Parameters local_file_name: amt\$local_file_name;
Local file name.

access_level: amt\$access_level;
Type of file I/O to be performed.

AMC\$RECORD

Record access.

AMC\$SEGMENT

Segment access (valid only for mass storage files whose file_organization attribute is not AMC\$INDEXED_SEQUENTIAL).

access_selections: amt\$file_access_selections;

Pointer to an array of one or more file attribute records. You must specify an attribute identifier and an attribute value in each record. The valid attributes are listed in table 6-2.

To specify no attribute values, specify a NIL pointer for the parameter or the keyword value NIL.

file_identifier: VAR of amt\$file_identifier;

File access identifier (used subsequently to reference this instance of open).

status: VAR of ost\$status;

Status record. The process identifier returned is either AA (for an indexed sequential file) or AM (AMC\$ACCESS_METHOD_ID).

**Condition
Identifiers**

ame\$attribute_validation_error
 ame\$concurrent_tape_limit
 ame\$file_not_known
 ame\$fo_access_level_conflict
 ame\$fo_device_class_conflict
 ame\$improper_access_level
 ame\$improper_append_open
 ame\$improper_fo_override
 ame\$improper_override_access
 ame\$improper_record_override
 ame\$improper_ss_block_override
 ame\$improper_us_block_override
 ame\$local_file_limit
 ame\$mbl_less_than_mibl
 ame\$mbl_less_than_mrl
 ame\$multiple_open_of_tape
 ame\$new_file_requires_append
 ame\$no_permission_for_access
 ame\$not_physical_access_device
 ame\$not_virtual_memory_device
 ame\$null_access_mode
 ame\$ring_validation_error
 ame\$terminal_task_limit
 ame\$unable_to_load_collate_tabl
 ame\$unable_to_load_error_exit
 ame\$unable_to_load_fap

For indexed sequential files only:

aae\$aam_requires_access
 aae\$adding_level_of_index
 aae\$altered_not_closed
 aae\$cant_open_new_an_old_file
 aae\$cant_open_old_a_new_file
 aae\$collated_key_needs_table
 aae\$data_pad_too_large
 aae\$file_reached_file_limit
 aae\$index_pad_too_large
 aae\$integer_key_gt_one_word
 aae\$key_length_0_or_undef
 aae\$max_rec_length_0_or_undef
 aae\$max_rec_length_too_big
 aae\$min_gt_max_record_length
 aae\$no_home_block_count
 aae\$rec_too_small_for_key

- Remarks** In preparing the file for I/O, the AMP\$OPEN procedure performs the following functions.
- Assigns a file identifier to this instance of open.
 - Registers the local file name of a new temporary file in the \$LOCAL catalog if no call or command has defined it previously.
 - Overrides previously defined temporary attribute values with any temporary attribute values specified by the access_selections parameter on the call.
 - For a new file, overrides previously defined structural attribute values with any corresponding attribute values specified by the access_selections parameter on the call. It then stores the new attribute values with the file.

For an old file, it compares the structural attribute values specified by the access_selections parameter with the structural attribute values stored with the file. If the values do not match, it returns abnormal status (AME\$ATTRIBUTE_VALIDATION_ERROR). For more information, see chapter 6, Defining File Attributes.
 - Prepares the instance of open for either segment access or record access according to the access_level parameter on the call.
 - Positions the file according to its open_position attribute.
 - Loads the file access procedure and error exit procedure if those attributes are defined.
 - Loads the collate table if the collate_table_name attribute is defined and the file is a new file. For an old file the collate table value saved from the original open of the file is made available; the table is not reloaded.

AMP\$CLOSE

Purpose Terminates access to a file for a specified instance of open.

Format AMP\$CLOSE (file_identifier, status)

Parameters **file_identifier**: amt\$file_identifier;
File identifier assigned by AMP\$OPEN.

status: VAR of ost\$status;
Status record. The process identifier is
AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_id
ame\$ring_validation_error
ame\$unrecovered_write_error

For indexed sequential files only:

aae\$delete_keys_this_open
aae\$get_keys_this_open
aae\$get_next_keys_this_open
aae\$last_error_repeated
aae\$put_keys_this_open
aae\$putreps_this_open
aae\$replace_keys_this_open

Remarks

- Closing a file terminates the association of the file_identifier parameter with an instance of open.
- If the file is an unlabeled tape file and the last operation to the file was an output operation, the procedure terminates the tape volume by writing two consecutive tapemarks on the file, and then positions the tape just prior to the two tapemarks.

Access Validation

When a task attempts to open a file, NOS/VE validates both the caller's ring number and the requested access modes. The valid open position depends on the access modes requested.

Ring Number Validation

The ring number of the caller is compared with the read, write, and execute ring attributes stored for the file. If the caller's ring is less than or equal to the ring attribute value, the requested access is granted.

Access Mode Validation

The access modes requested on the AMP\$OPEN call must be within the global access mode set. The global access mode set for a permanent file is specified when the file is attached to the job. The global access mode set for a temporary file always includes all access modes (read, append, modify, shorten, and execute).

For a new mass storage file, the access modes requested must include append.

Open Position for Appending

If an AMP\$OPEN call for an old file requests only append access, the open_position attribute value must be AMC\$OPEN_AT_EOI.

Implicit Release of File Data

If an AMP\$OPEN call for an old file requests append and shorten access with an open_position of AMC\$BOI, NOS/VE releases all existing data in the file. Although the file is empty, its space and file attribute set remain assigned to the file. For indexed sequential files, a message is issued to inform the user that this has happened.

Error Exit Procedure

Normally, a file interface procedure returns abnormal status directly to its caller. However, if an error exit procedure is defined for the specified instance of open, the file interface procedure passes the abnormal status to the error exit procedure. The status returned by the error exit procedure is the status returned to the caller. This allows the error exit procedure to perform error recovery for the instance of open.

A task can establish an error exit procedure to trap errors so that the task need not check for abnormal status after each file interface call during an instance of open. The error exit procedure is not effective for the AMP\$OPEN or AMP\$CLOSE calls.

Error Exit Procedure Attributes

The error_exit_name attribute can name an error exit procedure for a file. When the file is opened, the system searches for the procedure as an entry point in the task or as an entry point in the object library list. After finding the procedure, it loads the procedure in the program space, if it is not already loaded, and then stores the procedure address as a temporary attribute.

While the file is open, the program can change the error exit procedure used by replacing the address. It does so by calling AMP\$STORE with a pointer for the error_exit_procedure attribute. The new pointer address is used until it is replaced by another address or the file is closed. An address specified by AMP\$STORE is never preserved.

The procedure declaration of the error processing procedure must specify the XDCL attribute and have the parameter list defined as follows:

```
^procedure (file_identifier: amt$file_identifier;
           VAR status: ost$status)
```

The error processing procedure must be callable from the ring from which the file interface procedure is called.

When a file interface procedure (other than AMP\$OPEN or AMP\$CLOSE) returns an abnormal status record, the system checks the file attributes to determine whether the file has an error exit procedure. If it does, the system calls the procedure and passes it the file identifier and the abnormal status variable.

The procedure can then investigate the error and process it as desired. It could decide that the error can be ignored and change the status to normal; it could initiate recovery or diagnostic processing, or it could pass the same or different abnormal status condition to the file interface procedure, which, in turn, passes the condition to its caller.

File Sharing

Files assigned to the mass storage, interactive and null device classes can be shared among tasks; files assigned to tape cannot be shared. Sharing of interactive files is described in the Terminal Management chapter under Terminal Input and under Terminal Output.

Separate file positioning information is maintained for each instance of open that reads the file, including instances of open within the same task. However, all tasks having concurrent instances of open for a file cycle share a common end-of-information position.

Both temporary and permanent mass storage files and interactive files can be shared among tasks within a job. Only permanent mass storage files can be shared among tasks in different jobs.

All instances of open of a mass storage file share the same file copy in virtual memory. Each instance of open is constrained to the mode of access granted to it. A file operation performed by one instance of open is effective for all other instances of open. For example, an instance of open that can read the file can read data written by a concurrent instance of open.

More than one job can attach a permanent mass storage file at the same time if the share modes for the file allow it.

Reading a Shared File

Tasks within the same or different jobs that have opened the same file can read the file independently. A get call always uses the file positioning information maintained for its instance of open.

Writing a Shared File

In general, tasks writing to a shared file must coordinate their file access among themselves. If the tasks open the file for segment access, the coordinating information can be stored in the shared segment.

The system supports a simple form of write serialization for sequential files opened with the same local file name within the same job. In this case, each put call to the file uses the `global_file_address` and `global_file_position` attribute values to determine where it writes the record. These values are updated by each get or put call to the file. The shared values are available to each task that opens the file with that local file name within the same job.

Note that a get call does not use the `global_file_address` or `global_file_position` values to determine where to read; it always uses the values maintained for its instance of open. However, put calls do update the `global_file_address` and `global_file_position` values.

Retrieving Access Information

While a file is open, the system maintains access information for that instance of open. The access information items are listed in table 7-1.

The contents of the access information items change as the task performs I/O operations on the file. These operations are described in later chapters. A task can fetch the value of any access information item with an `AMP$FETCH_ACCESS_INFORMATION` call.

Table 7-1. File Information Record (AMT\$ACCESS_INFO)

Field	Content
item_returned	<p>Indicates whether the procedure returned a value for the item (boolean value).</p> <p>For indexed sequential files, AMP\$FETCH_ACCESS_INFORMATION always returns FALSE as the item_returned value for the following items.</p> <p style="padding-left: 40px;"> block_number current_byte_address previous_record_address previous_record_length volume_number volume_position </p>
key	Key field specifying the item to be returned in the record (AMT\$ACCESS_INFO_KEYS). The key identifier is the field name prefixed by AMC\$ (for example, AMC\$BLOCK_NUMBER for the block_number field).
block_number	Number of the last block accessed by record I/O (integer from 1 through AMC\$MAX_BLOCK_NUMBER).
current_byte_address	Current file position (byte offset into the file) (integer from 0 through AMC\$FILE_BYTE_LIMIT).
duplicate_value_inserted	Boolean indicating whether the last put or replace call wrote a record having an alternate key value that duplicates the alternate key value of a record already in the file (indexed sequential files only). The indicated duplication could be for any alternate key defined for the file.
eoi_byte_address	Current length of the file in bytes (integer from 0 through AMC\$FILE_BYTE_LIMIT).
error_count	Number of errors returned by file access requests for an indexed sequential file (integer from 0 through AMC\$MAX_ERROR_COUNT).

(Continued)

Table 7-1. File Information Record (AMT\$ACCESS_INFO)
(Continued)

Field	Content
error_status	Condition code returned as the status of the last file interface request for the file (OST\$STATUS_CONDITION).
file_position	Current file position of a file using record access (AMT\$FILE_POSITION). AMC\$BOI Beginning-of-information AMC\$BOP Beginning-of-partition AMC\$END_OF_KEY_LIST End of a key list in an alternate index. (See chapter 10 for more information.) AMC\$MID_RECORD Within a record AMC\$EOR End-of-record AMC\$EOP End-of-partition AMC\$EOI End-of-information
last_access_operation	Code indicating the latest access request issued for this instance of open (AMT\$LAST_ACCESS_OPERATION, integer from 105 through AMC\$MAX_OPERATION). The following lists the access requests and the corresponding constant identifier declarations. AMP\$ABANDON_KEY_DEFINITIONS amc\$abandon_key_definitions AMP\$APPLY_KEY_DEFINITIONS amc\$apply_key_definitions

(Continued)

Table 7-1. File Information Record (AMT\$ACCESS_INFO)
(Continued)

Field	Content
	AMP\$CLOSE
	amc\$close_req
	AMP\$CREATE_KEY_DEFINITION
	amc\$create_key_definition
	AMP\$DELETE
	amc\$delete_req
	AMP\$DELETE_KEY
	amc\$delete_key_req
	AMP\$DELETE_KEY_DEFINITION
	amc\$delete_key_definition
	AMP\$FETCH
	amc\$fetch_req
	AMP\$FLUSH
	amc\$flush_req
	AMP\$GET_DIRECT
	amc\$get_direct_req
	AMP\$GET_KEY
	amc\$get_key_req
	AMP\$GET_KEY_DEFINITIONS
	amc\$get_key_definitions
	AMP\$GET_NEXT
	amc\$get_next_req
	AMP\$GET_NEXT_KEY
	amc\$get_next_key_req
	AMP\$GET_NEXT_PRIMARY_KEY_LIST
	amc\$get_next_primary_key_list
	AMP\$GET_PARTIAL
	amc\$get_partial_req
	AMP\$GET_PRIMARY_KEY_COUNT
	amc\$get_primary_key_count
	AMP\$GET_SEGMENT_POINTER
	amc\$get_segment_pointer_req

(Continued)

Table 7-1. File Information Record (AMT\$ACCESS_INFO)
(Continued)

Field	Content
	AMP\$OPEN amc\$open_req
	AMP\$PUT_DIRECT amc\$put_direct_req
	AMP\$PUT_KEY amc\$put_key_req
	AMP\$PUT_NEXT amc\$put_next_req
	AMP\$PUT_PARTIAL amc\$put_partial_req
	AMP\$PUTREP amc\$putrep_req
	AMP\$REPLACE_KEY amc\$replace_key_req
	AMP\$REWIND amc\$rewind_req
	AMP\$SEEK_DIRECT amc\$seek_direct_req
	AMP\$SELECT_KEY amc\$select_key
	AMP\$SET_SEGMENT_EOI amc\$set_segment_eoi_req
	AMP\$SET_SEGMENT_POSITION amc\$set_segment_position_req
	AMP\$SKIP amc\$skip_req
	AMP\$START amc\$start_req
	AMP\$STORE amc\$store_req
	AMP\$WRITE_END_PARTITION amc\$write_end_partition_req

(Continued)

Table 7-1. File Information Record (AMT\$ACCESS_INFO)
(Continued)

Field	Content
	AMP\$WRITE_TAPE_MARK amc\$write_tape_mark_req
last_op_status	Indicates whether the last access request is active or complete. AMC\$ACTIVE Access request is active. AMC\$COMPLETE Access request is complete.
levels_of_indexing	Number of index levels in an indexed sequential file (integer from 0 through AMC\$MAX_INDEX_LEVEL). For more information, see chapter 10, Accessing Indexed Sequential Files.
previous_record_address	Starting address of the previous record (integer from 0 through AMC\$FILE_BYTE_LIMIT). It is valid only for files opened for record access. The value is defined only when the file position is AMC\$EOR.
previous_record_length	Number of bytes in the last full record accessed (integer from 0 through AMC\$MAXIMUM_RECORD). It is valid only for files opened for record access. The value is updated whenever the file position is AMC\$EOR, AMC\$BOP, or AMC\$EOP. For files accessed sequentially, the value is the length of the previous record (0 after an AMP\$WRITE_END_PARTITION call).
primary_key	Pointer to the location in which the primary key for the record at the current file position is returned. The pointer must be predefined before the AMP\$FETCH_ACCESS_INFORMATION call is made (indexed sequential files only).

(Continued)

Table 7-1. File Information Record (AMT\$ACCESS_INFO)
(Continued)

Field	Content
residual_skip_count	Number of units remaining to be skipped when the file delimiter which ended the skip was encountered (integer from 0 through AMC\$FILE_BYTE_LIMIT). The number of units requested minus the residual_skip_count yields the number of units skipped.
selected_key_name	Name of last key selected for the file (indexed sequential files only). If no alternate key has been selected, the name \$PRIMARY_KEY is returned.
volume_number	Number of the current tape volume in the volume sequence. The first volume in the sequence is volume 1 (integer from 1 through AMC\$MAX_VOLUME_NUMBER).
volume_position	Current position of the current tape volume. AMC\$BOV Beginning-of-volume. AMC\$AFTER_TAPEMARK After a tape mark. AMC\$EOV End-of-volume.

NOTE

The CYBIL declaration for AMT\$ACCESS_INFO in Appendix C lists additional fields besides those listed here. These additional fields are for features not yet implemented.

AMP\$FETCH_ACCESS_INFORMATION

Purpose Retrieves information about an open file.

NOTE

The information applies only to the specified instance of open.

Format AMP\$FETCH_ACCESS_INFORMATION (file_ identifier, access_information, status)

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

access_information: VAR of amt\$access_information;
File information array. Each record in the array specifies the access information item to be returned in the record. (See table 7-1.)

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_access_info_key
ame\$improper_file_id
ame\$ring_validation_error

For indexed sequential files only:
aae\$not_enough_permission

Accessing a File as a Memory Segment

8

CYBIL Data Storage	8-1
Virtual Memory Access	8-3
Segment Attributes	8-4
Segment Length	8-4
Segment Pointer	8-5
AMP\$GET_SEGMENT_POINTER	8-6
Cell Pointer	8-8
Heap Pointer	8-9
Allocating a Heap Within a Sequence	8-10
Sequence Pointer	8-13
Sharing a Segment Access File	8-15
Setting the End-of-Information Address	8-15
Setting the End-of-Information Address Using a	
Sequence Pointer	8-15
Setting the End-of-Information Address Using a Cell Pointer	8-17
Setting the Current Byte Address	8-17
AMP\$SET_SEGMENT_EOI	8-18
AMP\$SET_SEGMENT_POSITION	8-20



Accessing a File as a Memory Segment

8

NOS/VE provides two levels of access for mass storage files: record access and segment access. When a task opens a file, it specifies the access level for the instance of open.

NOTE

Segment access is valid only for mass storage files with sequential or byte addressable file organization. It is not valid for files with indexed sequential file organization or files assigned to the tape or terminal device classes; an attempt to get a segment pointer for a file assigned to the null device class returns a NIL pointer.

CYBIL Data Storage

When deciding whether to access a file as a segment, you should consider how segment access compares to the other data storage mechanisms available to a CYBIL program. A CYBIL program can use any or all of the following:

- The CYBIL run-time stack or default heap.
- Files read or written using segment access.
- Files read or written using record access.

When comparing use of the CYBIL run-time stack or default heap with use of a segment access file, consider the following:

- Segment access, the run-time stack, and the default heap all allow dynamic expansion of the task address space to fit task needs. (Space is allocated using the PUSH statement for the run-time stack, the ALLOCATE statement for the default heap, and ALLOCATE or NEXT statements for a segment access file.)
- Segment access, the run-time stack, and the default heap all allow you to read and write data using pointer variables declared within the task.
- Data stored in a segment access file can be accessed after the task terminates. Data stored in the run-time stack or default heap is discarded.

- Data stored in a segment access file is sharable. Data stored in the run-time stack or default heap is not sharable. (For an example of sharing a segment access file between tasks, see the queue communication example in the CYBIL System Interface manual.)
- Data stored in a sequence is contiguous whereas data stored in a heap is interspersed with system information. For example, data written as a sequence could later be read as an array because the data is contiguous. If the task cannot predict the required size of the sequence, it should write the sequence in a segment access file because a segment access file allows dynamic extension with contiguity of data.

When comparing use of segment access with use of record access, consider the following:

- A task that opens a file for record access reads and writes file data as records using the file interface calls described in chapters 9 and 10. A task that opens a file for segment access reads and writes file data using CYBIL statements.
- File I/O using segment access is more efficient than file I/O using record access because no explicit system calls are required to access data. Using segment access, the movement of data between memory and mass storage is done implicitly as the task references the data in memory.
- Unlike a record access file, a segment access file has no structure imposed on it by NOS/VE. Record and partition boundaries are not recognized. Opening a file for segment access allows the task to impose its own structure on the file data.
- Because NOS/VE imposes no structure on a segment access file, the task that writes data on the file is responsible for determining how the data can later be read. It should write data organization indicators as needed. A program that reads the file data must use the data conventions imposed by the program that wrote the data.
- Character data files to be referenced by NOS/VE commands should be read and written using record access.
- Error handling for a segment access file requires establishment of a condition handler for segment access conditions. (Condition handlers and segment access conditions are described in the CYBIL System Interface manual.)

Virtual Memory Access

When a task opens a file for segment access, the file is referenced as a segment of virtual memory. A virtual memory segment is a portion of the task's address space. Access to a segment is controlled by its access modes and ring attributes.

The system memory manager associates real pages of memory with the virtual memory segment. An address in a virtual memory segment is called a process virtual address (PVA). When the task references a PVA, the system memory manager associates the PVA with its real memory address.

The system memory manager ensures that all values written in memory are also stored in the mass storage copy of the file. Similarly, when data is read from a file, the system memory manager ensures that the referenced data is copied from the mass storage copy to memory.

Figure 8-1 illustrates the association of a PVA with a real memory address, that, in turn, is associated with a mass storage address. The system performs all translation of addresses; the process is transparent to the user.

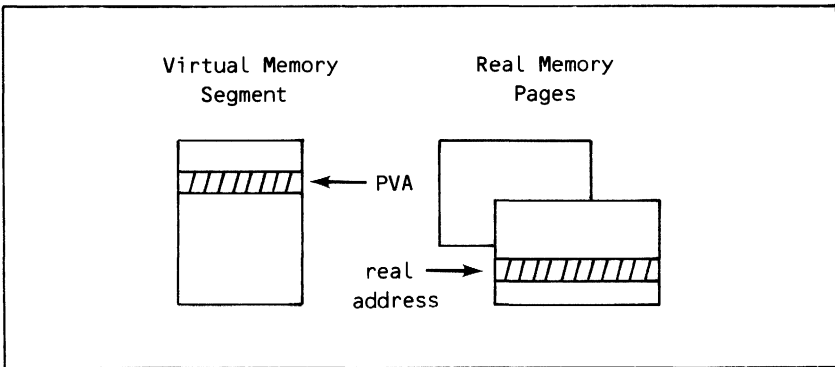


Figure 8-1. Virtual Address Translation

Segment Attributes

The ring attributes of the segment referenced as a segment access file are determined by the value of its file attribute `ring_attributes`. The access modes of the segment are determined by the value of its `access_modes` attribute. File attribute definition is described in chapter 6.

NOTE

If a file is opened for segment access without read access, any PACKED data structures written in the segment must have each component aligned on a byte boundary. To write a component of a packed data structure that is not aligned on a byte boundary, the system must perform a read operation before the write operation. The read operation requires read access. Therefore, if the file is opened without read access, an attempt to write an unaligned PACKED component causes an `access_fault` condition.

Segment Length

The maximum size of a segment access file is determined by the value of the `file_limit` file attribute. It is therefore recommended that you specify the `file_limit` attribute value when you open a file for segment access. In general, you do so by declaring a type and then specifying its size as the `file_limit` value.

For example, the following statement declares a heap type.

```
TYPE
    sequence_type = SEQ(REP 100 OF INTEGER);
```

The following statement declares an `access_selections` variable using the declared heap type.

```
VAR
    access_selections: [STATIC] ARRAY [1..1] OF
        amt$access_selections := [[amc$file_limit,
            (#SIZE(sequence_type))]];
```

The following AMP\$OPEN call references the `access_selections` variable.

```
amp$open(lfn, amc$segment, ^access_selections,
    fid, status);
```

Segment Pointer

The CYBIL statements (such as `ALLOCATE` and `NEXT`) that reference the segment require a pointer to the segment. To get the pointer, the task calls `AMP$GET_SEGMENT_POINTER`.

The `AMP$GET_SEGMENT_POINTER` call specifies the type of pointer required and, therefore, the data storage type accessed through the segment pointer. The call can request a cell pointer, a sequence pointer, or a heap pointer.

AMP\$GET_SEGMENT_POINTER

Purpose Returns a pointer to the virtual memory segment assigned to a file.

Format AMP\$GET_SEGMENT_POINTER (**file_identifier**, **pointer_kind**, **segment_pointer**, **status**)

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

pointer_kind: amt\$pointer_kind;
Type of pointer to be returned.

AMC\$CELL_POINTER

Pointer to a cell.

AMC\$HEAP_POINTER

Pointer to an adaptable heap.

AMC\$SEQUENCE_POINTER

Pointer to an adaptable sequence.

segment_pointer: VAR of amt\$segment_pointer;
Record containing the pointer kind identifier and a pointer variable initialized by the call.

kind

Key field determining the pointer type returned.

AMC\$CELL_POINTER

Cell pointer returned in the cell_pointer field.

AMC\$HEAP_POINTER

Heap pointer returned in the heap_pointer field.

AMC\$SEQUENCE_POINTER

Sequence pointer returned in the sequence_pointer field.

cell_pointer

Cell pointer (^CELL).

heap_pointer

Adaptable heap pointer [^HEAP(*)].

sequence_pointer

Adaptable sequence pointer [^SEQ(*)].

status: VAR of ost\$status;
 Status record. The process identifier is
 AMC\$ACCESS_METHOD_ID.

**Condition
 Identifiers**

ame\$improper_file_id
 ame\$improper_pointer_kind
 ame\$read_of_empty_segment
 ame\$ring_validation_error
 ame\$write_of_empty_segment

Remarks

- If the pointer returned is a cell pointer, the call initializes the byte offset portion of the pointer to the current_byte_address of the file.
- If the pointer returned is an adaptable heap pointer, the call initializes the byte offset portion of the pointer to zero, the address of the first byte of the segment. It initializes the heap length portion of the pointer depending on the content and access modes of the file as follows:
 - If the file contains no data (null length) and the task has requested append access to the file, the heap length is initialized to the file_limit attribute value. (If the file contains no data and the task has not requested append access, the call returns abnormal status.)
 - If the file contains data but the task has not requested append access to the file, the heap length is initialized to the eoi_byte_address of the file.
 - If the file contains data and the task has requested append access to the file, the heap length is initialized to the file_limit attribute value.
- If the pointer returned is an adaptable sequence pointer, the call initializes the byte offset in the pointer to the first byte of the segment and the current position in the pointer to the current_byte_address value. It initializes the sequence length in the pointer the same way it initializes the heap length in an adaptable heap pointer.
- An AMP\$GET_SEGMENT_POINTER call does not change the contents of the file.
- An AMP\$GET_SEGMENT_POINTER call that specifies a file assigned to the null device class (such as the \$NULL file) returns a NIL pointer.

Cell Pointer

A cell pointer is the process virtual address (PVA) of a location in virtual memory. A PVA contains a ring number, the segment number, and a byte offset within the segment. When `AMP$GET_SEGMENT_POINTER` returns a cell pointer to the segment, it initializes the PVA byte offset to the `current_byte_address` of the file. (An `AMP$FETCH_ACCESS_INFORMATION` call can return the `current_byte_address` value.)

For example, if the file is opened at its beginning-of-information, the byte offset is set to 0, pointing to the first byte in the segment. However, if the file is opened at its end-of-information, the byte offset is set to the `eof_byte_address` and the file is positioned for appending data.

A cell pointer can only be used within an assignment statement; it cannot be used to reference the segment directly. To use the cell pointer, the task must perform the following steps.

1. Declare a pointer to a fixed type data structure. For example, the following statement declares a pointer to a character array:

```
VAR
    array_pointer: ^ARRAY[1..1000] OF char;
```

2. Assign the cell pointer value to the declared pointer variable. For example, the following statement assigns the value of a cell pointer returned by `AMP$GET_SEGMENT_POINTER` to the previously declared pointer variable:

```
array_pointer := segment_pointer.cell_pointer;
```

3. Dereference the pointer variable to reference space in the segment. For example, the following statement stores 'A' as the first character in the array:

```
array_pointer^[1] := 'A';
```

Heap Pointer

A heap pointer contains a PVA and the heap length. `AMP$GET_SEGMENT_POINTER` initializes the byte offset of a heap pointer to zero, pointing to the first byte in the heap.

The heap length initialization depends on whether the file is opened for append access. If the file is opened for append access, `AMP$GET_SEGMENT_POINTER` initializes the heap length to the `file_limit` value, the maximum length of the file. If the file is not opened for append access, `AMP$GET_SEGMENT_POINTER` initializes the heap length to the `eof_byte_address`, its current end-of-information.

Before executing the first `ALLOCATE` statement for a new heap, the task must execute a `RESET` statement to ensure that the heap is initialized. It can then execute `ALLOCATE` statements to reserve space for variables in the heap.

The `ALLOCATE` statement returns a `NIL` pointer if the heap does not contain enough free space for the variable. The task should check for a `NIL` pointer after each `ALLOCATE` statement. If the task attempts to dereference a `NIL` pointer, either a `CYBIL` run-time error or a segment access condition occurs, depending on whether the program compilation requested `NIL` pointer checking (`RUN_CHECKS=N` on the `CYBIL` command).

The task can free space within the heap with a `FREE` statement. The `FREE` statement specifies a pointer to the variable whose space is to be released. A `RESET` statement frees all space in the heap.

Allocating a Heap Within a Sequence

Data written in a heap is more easily accessed by subsequent tasks if the heap is created within a sequence. The sequence could begin with a directory to the heap variables followed by the heap itself.

NOTE

To be used by a subsequent task, a directory of pointers must contain CYBIL relative pointers. Absolute pointers are not usable because they include the segment number, which could differ for the next instance of open.

For example, figure 8-2 lists a program that performs the following steps.

1. Opens a file for segment access and gets a sequence pointer to the file.
2. Reserves space for a heap directory and a heap in the sequence.
3. Reserves space for an integer variable in the heap and then stores a value in the heap variable and a relative pointer to the heap variable in the directory.
4. Closes the file.


```

MODULE segment_example;
*copyc amp$open
*copyc pmp$exit
*copyc amp$get_segment_pointer
*copyc amp$close

PROGRAM heap_in_sequence;

CONST
{This is the number of integer variables in the}
{heap.}
    number_of_variables = 1;

TYPE
    heap_type = HEAP(REP number_of_variables
        OF integer),
    relative_pointer_type = REL(heap_type) ^integer,
    directory_type = ARRAY [1..number_of_variables] OF
        relative_pointer_type;

VAR
    lfn: [STATIC] amt$local_file_name := 'FILE1',
    status: ost$status,
    fid: amt$file_identifier,
    segment_pointer: amt$segment_pointer,

{The following specifies the segment length as }
{the size of the directory plus the size of }
{the heap.}
    access_selections: [STATIC] ARRAY [1..1] OF
        amt$access_selection := [[amc$file_limit,
            (#SIZE(directory_type) + #SIZE(heap_type))]],

    directory_pointer: ^directory_type,
    heap_pointer: ^heap_type,
    variable_pointer: ^integer;

```

*(Continued)***Figure 8-2. Example of Allocating a Heap in a Sequence**

(Continued)

```

amp$open (lfn, amc$segment, ^access_selections,
          fid, status);
IF NOT status.normal THEN
  pmp$exit (status);
IFEND;

amp$get_segment_pointer (fid, amc$sequence_pointer,
                        segment_pointer, status);
IF NOT status.normal THEN
  pmp$exit (status);
IFEND;

RESET segment_pointer.sequence_pointer;
NEXT directory_pointer IN
  segment_pointer.sequence_pointer;
IF directory_pointer < > NIL THEN
  NEXT heap_pointer IN segment_pointer.sequence_pointer;
  IF heap_pointer < > NIL THEN
    RESET heap_pointer^;
    ALLOCATE variable_pointer IN heap_pointer^;
    IF variable_pointer < > NIL THEN
      variable_pointer^ := 1;
      directory_pointer^[1] :=
        #REL(variable_pointer, heap_pointer^);
    IFEND;
  IFEND;
IFEND;

amp$close (fid, status);
IF NOT status.normal THEN
  pmp$exit (status);
IFEND;

PROCEND heap_in_sequence;
MODEND segment_example;

```

Figure 8-2. Example of Allocating a Heap in a Sequence

After the segment access file is written using the program listed in figure 8-2, another task can read the value written by using the same variable declarations used by the task that wrote the file. The task would open the file for segment access, get a sequence pointer to the segment, and then execute the following statements to read the value from the heap:

```

RESET segment_pointer.sequence_pointer;
NEXT directory_pointer IN
    segment_pointer.sequence_pointer;
IF directory_pointer < > NIL THEN
    NEXT heap_pointer IN
        segment_pointer.sequence_pointer;
    IF heap_pointer < > NIL THEN
        RESET heap_pointer^;
        variable_pointer :=
            #PTR(directory_pointer^[1], heap_pointer^);
        IF variable_pointer < > NIL THEN
            value := variable_pointer^;
        IFEND;
    IFEND;
IFEND;

```

The #PTR function returns a pointer to the integer value in the heap. The pointer is then dereferenced to assign the integer value to the integer variable, VALUE.

Sequence Pointer

A sequence pointer has three components: a pointer to the beginning of the sequence, the current position in the sequence, and the sequence length. The values of the beginning-of-sequence pointer and the sequence length do not change during the lifetime of the pointer; the value of the current position is changed by each NEXT or RESET statement.

AMP\$GET_SEGMENT_POINTER initializes the pointer to the beginning of the sequence to zero. It initializes the current position to the current_byte_address. The initialization of the sequence length depends on whether the file is opened for append access.

If the file is opened for append access, AMP\$GET_SEGMENT_POINTER initializes the sequence length to the file_limit value; in this case, the sequence length is the maximum length of the file. If the file is not opened for append access, AMP\$GET_SEGMENT_POINTER initializes the sequence length to the eoi_byte_address; the sequence length is the current length of the file.

A NEXT statement is used to reserve space for a variable in the sequence beginning at the current position. As described in the CYBIL Language Definition manual, the NEXT statement has the following format:

NEXT pointer_variable IN sequence_pointer;

As shown, the NEXT statement specifies two pointers: a pointer variable and the sequence pointer. The pointer variable type determines the amount of space reserved by the NEXT statement.

A NEXT statement initializes the byte offset in the pointer variable to the current position value in the sequence pointer. It then advances the current position to the next available byte in the sequence. For example, if the current position in the sequence pointer is 0 before the NEXT statement and the variable to be reserved by the NEXT statement is 10 bytes long, the NEXT statement initializes the pointer variable to point to byte 0 in the sequence and advances the current position in the sequence pointer to 10. A second execution of the same NEXT statement would set the pointer variable to byte 10 and the current position in the sequence pointer to byte 20.

Subsequent NEXT statements continue to advance the current position within the segment pointer. When a NEXT statement returns a NIL pointer, the current position has reached the sequence length, and no more space can be reserved.

The task should check for a NIL pointer after each NEXT statement. If the task attempts to dereference a NIL pointer (pointer[^]), either a CYBIL runtime error or a segment access condition occurs depending upon whether the program compilation requested CYBIL NIL pointer checking (RUNTIME_CHECKS=N on the CYBIL command).

A RESET statement can always reset the current position to the beginning of the sequence or to a previous position in the sequence (if the pointer value to the position has not been discarded).

The RESET and NEXT statements do not read data from or write data to the file. Data is read or written by dereferencing a variable pointing to space reserved in the sequence.

The RESET and NEXT statements do not change the current_byte_address or the eoi_byte_address of the file. To change the current_byte_address or the eoi_byte_address, you call AMP\$SET_SEGMENT_POSITION or AMP\$SET_SEGMENT_EOI, respectively.

Sharing a Segment Access File

When a task writes data in a segment access file for use by another instance of open, it can store the following information for later use:

- The position where the data ends (its `eoi_byte_address`).
- The position within the file to which the next segment pointer should be initialized (its `current_byte_address`), assuming the file is opened without repositioning (open_position attribute value `AMC$OPEN_NO_POSITIONING`).

Setting the End-of-Information Address

If the task that writes data in a segment access file does not set the `eoi_byte_address` of the value, the system assumes the file extends to the end of the highest page referenced by the task. However, if the file data does not extend to the end of the highest page referenced, the file includes unused space (which might be considered data by the next reader of the file). To reduce the file length to include only the space used, the task calls `AMP$SET_SEGMENT_EOI` to store the `eoi_byte_address` of the file.

A task cannot pass a heap pointer to `AMP$SET_SEGMENT_EOI`. It can only pass a cell pointer or a sequence pointer.

Setting the End-of-Information Address Using a Sequence Pointer

If the segment pointer is a sequence pointer, the task specifies on the `AMP$SET_SEGMENT_EOI` call the segment pointer value returned by the `NEXT` statement for the last element in the sequence. For example, if the end-of-information is to be after the tenth element in the sequence, the task should pass the value of the segment pointer after the tenth `NEXT` statement is executed.

When setting the `eo_i_byte_address` of a sequence, the task could set the end-of-information at a previous element in the sequence. For example, suppose a task uses the following statements to reserve space for two integer variables in a sequence and assign values to the variables.

```

NEXT variable1_pointer IN
    segment_pointer.sequence_pointer;
IF variable1_pointer < > NIL THEN
    variable1_pointer^ := 1;
NEXT variable2_pointer IN
    segment_pointer.sequence_pointer;
IF variable2_pointer < > NIL THEN
    variable2_pointer^ := 2;
IFEND;
IFEND;

```

The task then decides to discard the second variable. To do so, it uses the following statements:

```

RESET segment_pointer.sequence_pointer TO
    variable2_pointer;
amp$set_segment_eoi (fid, segment_pointer, status);

```

The `AMP$SET_SEGMENT_EOI` call sets the end-of-information after the first variable in the sequence.

Setting the End-of-Information Address Using a Cell Pointer

If the segment pointer is a cell pointer, the task sets the end of information address by initializing the cell pointer to the byte following the data storage used.

For example, suppose the cell pointer was assigned to point to an array of 500 elements. Assuming that the task stored data in the first 425 elements of the array, it should set the end of information at the next element. Therefore, before calling `AMP$SET_SEGMENT_EOI`, the task initializes the cell pointer as follows:

```
segment_pointer.cell_pointer := ^array_pointer^[426];
```

It then specifies the segment pointer on an `AMP$SET_SEGMENT_EOI` call.

Setting the Current Byte Address

As stated before, when `AMP$GET_SEGMENT_POINTER` returns a segment pointer, it initializes the byte offset in the pointer to the `current_byte_address` of the file. The `current_byte_address` where the next segment pointer should be initialized can be set by an `AMP$SET_SEGMENT_POSITION` call.

If the file is opened without repositioning (`open_position` attribute value `AMC$OPEN_NO_POSITIONING`), the `current_byte_address` provided by a previous file accessor is used in the segment pointer. This is so only if the file is opened without repositioning; opening at the beginning of information (`AMC$OPEN_AT_BOI`) or end-of-information (`AMC$OPEN_AT_EOI`) changes the `current_byte_address`.

AMP\$SET_SEGMENT_EOI

Purpose Sets the byte address of the end of information (EOI) of a file. It also sets the `current_byte_address` and the `global_file_address` to the new EOI address.

NOTE

To lengthen the file, the instance of open must have append access. To shorten the file, the instance of open must have shorten access.

Format **AMP\$SET_SEGMENT_EOI (file_identifier, segment_pointer, status)**

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

segment_pointer: amt\$segment_pointer;
Segment pointer to the new end-of-information of the file. The pointer can contain a cell pointer or a sequence pointer; it cannot contain a heap pointer.

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_id
ame\$improper_segment_number
ame\$improper_segment_pointer
ame\$ring_validation_error
ame\$set_eoi_needs_append
ame\$set_eoi_needs_shorten
ame\$set_on_adaptable_heap

Remarks

- The procedure uses the byte offset in the segment pointer as the new `eo_i_byte_address` value. Assuming that the next instance of `open` does not request append access, the file length extends to the stored `eo_i_byte_address`.
- Calling the `AMP$SET_SEGMENT_EOI` procedure stores the actual end-of-information of the file. Otherwise, if the `AMP$SET_SEGMENT_EOI` procedure is not called, the end-of-information is assumed to be the first byte beyond the end of the highest page referenced. The last page of the file could be only partially filled with data. Therefore, subsequent tasks could read file space that contains invalid information.
- To shorten the file (assuming the task has shorten privilege to the file), the call specifies a new EOI address numerically less than the former EOI address. The procedure discards the pages following the new EOI address. However, it retains the page containing the new EOI address; all data within that page remains available.
- The segment pointer specified cannot be a heap pointer. The EOI of an adaptable CYBIL heap is always extended to the end of the highest page the task has referenced in the heap.
- A task can determine the current end-of-information byte address by calling `AMP$FETCH_ACCESS_INFORMATION`.

AMP\$SET_SEGMENT_POSITION

Purpose Sets the current byte address and global file address of a file.

Format AMP\$SET_SEGMENT_POSITION (**file_identifier**, **segment_pointer**, **status**)

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

segment_pointer: amt\$segment_pointer;
Segment pointer to the new current byte address. The pointer can contain a cell pointer or a sequence pointer; it cannot contain a heap pointer.

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_id
ame\$improper_segment_number
ame\$improper_segment_pointer
ame\$ring_validation_error
ame\$set_on_adaptable_heap
ame\$set_pos_beyond_eoi

- Remarks:**
- When creating a new file, a task must call AMP\$SET_SEGMENT_EOI before calling AMP\$SET_SEGMENT_POSITION.
 - The segment pointer specified cannot be a heap pointer because a heap does not have a position.
 - By storing the current position of the file, the next task can open the file at that position by specifying no positioning on the next AMP\$OPEN call.
 - A task can determine the current byte address by calling AMP\$FETCH_ACCESS_INFORMATION.

Accessing Sequential and Byte Addressable Files

Logical File Structure	9-1
Working Storage Area	9-1
Record Types	9-2
CDC Variable (V) Record Type	9-2
ANSI Fixed-Length (F) Record Type	9-2
Undefined (U) Record Type	9-3
File Blocking	9-4
System-Specified Blocking	9-4
U Record Type	9-4
User-Specified Blocking	9-4
V Record Type	9-5
F Record Type	9-5
U Record Type	9-6
Sequential Record Access	9-7
Using Sequential Access Calls to Write a Byte Addressable File	9-8
Random Record Access	9-9
File Directory Use	9-10
User-Specified Blocking	9-10
AMP\$SEEK_DIRECT	9-11
Byte Addressable File Example	9-12
File Positioning	9-14
Positioning a File by Records or Partitions	9-14
File Position After a Skip	9-14
Skip That Encounters File Boundaries	9-16
AMP\$REWIND	9-17
AMP\$SKIP	9-18
Reading Records	9-20
File Position Returned	9-20
Get Calls	9-20
AMP\$GET_DIRECT	9-21
AMP\$GET_NEXT	9-23
AMP\$GET_PARTIAL	9-26
Writing Records	9-29
Establishing a New End-of-Information	9-29
Padding Fixed-Length Records	9-29
Truncating Fixed-Length Records	9-30
Writing Records Longer Than the Working Storage Area	9-30
AMP\$FLUSH	9-31
AMP\$PUT_DIRECT	9-33
AMP\$PUT_NEXT	9-35
AMP\$PUT_PARTIAL	9-37
Writing Partition Delimiters	9-38
AMP\$WRITE_END_PARTITION	9-39



Accessing Sequential and Byte Addressable Files

9

Opening a file for record access indicates that get and put calls are used to read and write records of data. To open a file for record access, specify `AMC$RECORD` for the `access_level` parameter on the `AMP$OPEN` call.

Record access is valid for all device classes. (Segment access is valid only for mass storage files.)

Record access offers device class independence. The same get or put call can read or write a record to any device class.

Logical File Structure

Opening a file for record access indicates that the program gets and puts file data within a file structure. File data exists within records. Records can be grouped into partitions if the record type used has partition delimiters.

The beginning-of-information for record access is the point at which the system can begin to read the first record. The end-of-information is the point immediately after the last record in the file.

The CDC variable (V) record type is the only record type that supports file partitioning. A file that uses the V record type consists of one or more partitions. If the file contains no end-of-partition delimiters, the entire file is one partition. A partition delimiter is a special record separating two partitions.

Working Storage Area

Record access calls specify a working storage area. When putting data in the file, the system copies data from the working storage area to a buffer it maintains. It manages the writing of data from the buffer to the file. When getting data from the file, the system reads data from the file to its buffer. It then copies data from the buffer to the working storage area.

Record Types

When a file is opened for record access, the `record_type` file attribute determines the record format that the system reads and writes on the file. The record types are CDC variable (V), ANSI fixed length (F), and undefined (U).

CDC Variable (V) Record Type

The V record type has the following characteristics.

- Default record type for NOS/VE.
- Supports fixed or variable record lengths.
- Supports partial record I/O and file partitioning.

Each V record has a record header. The header contains the record length and the length of the preceding record.

The end-of-partition delimiter for the V record type is a record header that has a record length of zero and its end-of-partition flag set.

The system writes the header when it writes the record. It uses the header information for positioning of the file. When reading a record, it does not copy the record header to the working storage area.

ANSI Fixed-Length (F) Record Type

The F record type has the following characteristics:

- Supports data interchange between differing systems because it is an ANSI standard record type.
- Supports partial record I/O, but does not support partitioning.
- Provides efficient storage of records of constant length. If a record is shorter than the fixed record length, however, the system pads the record to the fixed record length.

The fixed record length is the number of bytes specified by the `max_record_length` attribute value. Depending on the amount of file space used for record padding, the V record type is usually more space efficient for variable length records.

If the `block_type` is user-specified and the block being written does not have space for another record, the system pads the block with circumflex (^) characters. The user-specified minimum block length must be at least the `max_record_length` value. For more information, see User-Specified Blocking in this chapter.

Undefined (U) Record Type

The system considers a file with U record type as an unstructured byte string. The U record type has the following characteristics:

- Supports tape files for which a block is equivalent to a record.
- Supports data interchange with differing systems without using an ANSI standard record type.
- Supports partial record I/O but does not support file partitioning.
- Any file can be read as U records regardless of its previous record type.

The task specifies the starting location and length of each record. The system imposes no structure on the file other than file blocking.

If the `block_type` is user-specified and the record being written is shorter than the `min_block_length` attribute value, the system pads the block with circumflex (^) characters. For more information, see User-Specified Blocking in this chapter.

File Blocking

Data within a tape or mass storage file is written and read as a series of blocks. NOS/VE supports both system-specified and user-specified file blocking. The `block_type` file attribute specifies the physical group of records in the file.

System-Specified Blocking

In system-specified blocking, the system determines the size of file blocks according to the storage device on which the file resides. Mass storage blocks are 4,096 bytes or less (depending on the device); tape blocks are 4,128 bytes.

In system-specified blocking, records are always contiguous because blocks are not padded. Records can span system-specified blocks. The `max_block_length` and `min_block_length` attributes have no effect on system-specified blocking.

Generally, system-specified blocking is invisible to the user program.

U Record Type

U records can span system-specified blocks. Unlike other record types, the system cannot determine where a U record begins or ends. Therefore, the task is responsible for managing the location and length of each record in the file.

With U record type and system-specified blocking, processing of the get calls changes as follows.

- An `AMP$GET_NEXT` call always returns end-of-record as its file position unless the end-of-information was encountered.
- An `AMP$GET_PARTIAL` call always returns mid-record as its file position if data was transferred by the call.
- The `skip_option` parameter on an `AMP$GET_PARTIAL` call is ignored.
- `AMP$GET_PARTIAL` calls do not accumulate the record length. The `record_length` parameter is undefined after the call.

User-Specified Blocking

In user-specified blocking, the `max_block_length` and `min_block_length` attributes determine the range of block sizes written to a device. If a block is shorter than the size specified by the `min_block_length` attribute, the block is padded with the circumflex (^) character.

A file with user-specified blocking must be initially created and appended sequentially.

User-specified blocking differs for each record type.

V Record Type

With V record type and user-specified blocking, the length of each file block is between the `max_block_length` and `min_block_length` values. Although the data in a V record can span blocks, the V record header cannot. Therefore, the block length varies, as necessary, to accommodate record headers.

F Record Type

With F record type and user-specified blocking, an integral number of records is included in each block. The block length is between the `min_block_length` and `max_block_length` attribute values. An F record cannot span a user-specified block.

For example, if the `min_block_length` value is 25 bytes, the `max_block_length` value is 4,128 bytes, and the `max_record_length` value (the fixed record length) is 100 bytes, the system writes the file with 40 records (4,000 bytes) per block because 40 is the maximum number of records that can be written without exceeding the `max_block_length` value.

NOTE

When using F record type and user-specified blocking, the fixed record length specified by the `max_record_length` attribute should be greater than the `min_block_length` value for the file. If it is not, the system pads each block with circumflex (^) characters. For example, if the `min_block_length` is 50 bytes and the fixed record length is 30 bytes, each block would consist of one 30-byte record followed by 20 bytes of block padding.

U Record Type

With U record type and user-specified blocking, the system processes each block as a record. A U record cannot span a user-specified block.

A block boundary is recognized as a record boundary. A put call (or a series of put_partial calls) that writes a record writes one block. The working storage length on the put call (or the cumulative record length of the series of put_partial calls) specifies the size of the block written.

A get call to read a record reads one block. If the working storage length on the get call is less than the block length, the call returns AMC\$MID_RECORD as the file position.

NOTE

When using U record type and user-specified blocking, the length of each record written should be greater than the min_block_length value for the file. If it is not, the system pads the block containing the short record with circumflex (^) characters. This could result in the reading of file data suffixed by circumflex characters. For example, if the min_block_length of a file is 20 bytes and a put call writes a 15-byte record to the file, the system would write five additional circumflex characters to pad the block to its minimum length. A get call to read the block would read the circumflex characters with the file data.

Sequential Record Access

If the `file_organization` attribute of a file is `AMC$SEQUENTIAL`, only sequential access calls can get and put data on the file.

The sequential access calls do not specify a byte address where the system is to get or put data. Sequential access calls always get or put data at the current file position.

The following are the sequential access calls:

`AMP$GET_NEXT`

Gets a complete record.

`AMP$GET_PARTIAL`

Gets a complete record or part of a record.

`AMP$PUT_NEXT`

Puts a complete record.

`AMP$PUT_PARTIAL`

Puts a complete record or part of a record.

A program can get any existing record on a sequential file by positioning the file at the beginning of the record.

By positioning the file at its end-of-information, a program can append records in sequence.

Doing a GET or PUT operation positions the file at the end of a record.

If a program overwrites an existing record, all data following the record is lost. After completion of a put operation to a sequential file, the system changes the end-of-information of the file to the current position of the file. The file position after a put operation is immediately after the data just copied to the buffer. Therefore, a program cannot get file data after the most recently written record because the system cannot get data after the end-of-information of the file. It is recommended that the `file_position` be checked after each GET call.

Sequential access calls are device-independent. Programs use the same calls to get and put data on any device class. This is possible because the system performs the physical access operations between the device and its buffer space in central memory.

Sequential access calls use file space efficiently because the calls get and put contiguous records. Random access calls can leave unused space between records.

Using Sequential Access Calls to Write a Byte Addressable File

Sequential access calls can write a file whose file organization attribute is byte addressable. Each put call returns the byte address of the record it writes. While writing the file, the task stores the byte address of each record in a directory. After the file is written, a task can use random access calls and the addresses in the directory to read records.

Random Record Access

If the file `_organization` attribute of a file is `AMC$BYTE_ADDRESSABLE`, the system gets or puts data at the specified byte address.

The bytes in a file are numbered consecutively beginning with 0; a byte address is the number of a byte in the file.

A program can explicitly or implicitly change the current byte address. A random access call explicitly changes the current byte address; any call that gets or puts data to a byte addressable file implicitly changes the current byte address.

The following are the random access calls.

`AMP$GET_DIRECT`

Gets a record at the specified byte address.

`AMP$PUT_DIRECT`

Puts a record at the specified byte address.

`AMP$SEEK_DIRECT`

Changes the current byte address to a specified value.

Both random access calls and sequential access calls can get and put data on a byte addressable file. If the first call is a random access call specifying the byte address to be accessed, subsequent calls can be sequential access calls that get or put data from the byte address implicitly set by the random access call. For example, if the first call specifies byte address 200 and gets 100 bytes, a subsequent sequential get call gets data beginning at byte address 300.

If the blocking type is system-specified and append access is specified for the instance of open, a call to put data at a byte address past the end-of-information extends the file past that address up to the file limit. A call to get data at a byte address past the end-of-information returns abnormal status.

If the blocking type is system-specified, and append is specified as the access mode for the instance of open, an `AMP$PUT_DIRECT` call that specifies a byte address beyond the end of the file extends the file to include the space for the record written. If not previously written, the space between the former end-of-information and the new record is then initialized. Creating unused spaces within the file in this manner is not recommended, especially if variable length records are used. It is not allowed for user-specified blocking.

File Directory Use

To access a record randomly, you must know its byte address in the file. If the file was written sequentially with F record type and system-specified blocking, the byte address of each record can be computed. Otherwise, the byte address of each record must be recorded in a directory.

The directory must be modified as the records in the file are created. Each put call returns the byte address of the record written; the task must record the byte address of the record in the task's directory. The task must fetch the byte address value from the directory before issuing a call to access a record randomly. The location and management of the directory is the responsibility of the task.

User-Specified Blocking

If a file is created with user-specified blocking, it must be created contiguously.

Random calls can read and modify the records.

NOTE

When rewriting variable-length records, you must ensure that the new record is the same length as the record overwritten in order to maintain record integrity. The system does not check that the record lengths are the same.

AMP\$SEEK_DIRECT

Purpose Sets the current byte address of a byte addressable file.

NOTE

The `access_level` file attribute must be `AMC$RECORD` and the `file_organization` attribute must be `AMC$BYTE_ADDRESSABLE`.

Format **AMP\$SEEK_DIRECT (file_identifier, byte_address, status)**

Parameters

file_identifier: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

byte_address: amt\$file_byte_address;
New byte address of the file (integer 0 through AMC\$FILE_BYTE_LIMIT).

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers

ame\$conflicting_access_level
ame\$file_organization_conflict
ame\$improper_access_attempt
ame\$improper_file_id
ame\$position_beyond_eoi
ame\$position_beyond_file_limit
ame\$ring_validation_error

Remarks

- By calling AMP\$SEEK_DIRECT, you can use sequential access calls (AMP\$GET_NEXT, AMP\$GET_PARTIAL, AMP\$PUT_NEXT, and AMP\$PUT_PARTIAL) to read a byte addressable file. After an AMP\$SEEK_DIRECT call sets the current byte address, the subsequent sequential access call begins reading or writing data at that address.
- The task must position the file to the beginning of a record.

Byte Addressable File Example

The following program example writes a byte addressable file using sequential access calls. The byte address of each record written is stored in a directory. After the file is written, the program writes the directory at the beginning of the file.

```

MODULE byte_address_example;

?? PUSH (LISTEXT := ON) ??
*copyc amp$open
*copyc amp$seek_direct
*copyc amp$put_next
*copyc amp$put_direct
*copyc amp$close
?? POP ??

PROCEDURE byte_addressable
  (lfn: amt$local_file_name;
   VAR status: ost$status);

CONST
  max_index = 100;

VAR
  access_selections: amt$file_access_selections,
  fid: amt$file_identifier,
  address: amt$file_byte_address,

  record_ptr: ^RECORD
    link: amt$file_byte_address,
    number: integer
  RECEND,

  directory_ptr: ^ARRAY [1 .. max_index] OF
    amt$file_byte_address,
  index: 1 .. max_index;

{Specifies byte_addressable file organization}
PUSH access_selections: [1..1];
access_selections^[1].key := amc$file_organization;
access_selections^[1].file_organization :=
  amc$byte_addressable;

```



```

AMP$OPEN (lfn, amc$record, access_selections, fid,
          status);
IF NOT status.normal THEN
  RETURN;
IFEND;

PUSH directory_ptr;
PUSH record_ptr;

{ Sets the current byte address so that space is }
{ left at the beginning of the file for the address }
{ directory.}

AMP$SEEK_DIRECT (fid, (#SIZE(directory_ptr^) + 1),
                status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Initializes first record to be written.}

record_ptr^.link := 0;
record_ptr^.number := 1;

{ Writes a sequence of records }

FOR index := 1 TO max_index DO
  AMP$PUT_NEXT (fid, record_ptr, #SIZE(record_ptr^),
               directory_ptr^[index], status);
  record_ptr^.link := directory_ptr^[index];
  record_ptr^.number := record_ptr^.number + 1;
FOREND;

{ Writes the address directory at the beginning of }
{ the file.}

AMP$PUT_DIRECT (fid, directory_ptr,
                #SIZE(directory_ptr^), 0, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

AMP$CLOSE (fid, status);
PROCEND byte_addressable;
MODEND byte_address_example;

```

File Positioning

The initial position of a file is specified by its `open_position` attribute. To change the file position, a program can get or put data or issue file positioning calls. The following are the file positioning calls:

AMP\$REWIND

Rewinds the file so that it is positioned at its beginning.

AMP\$SKIP

Repositions the file forward or backward. The call specifies the number of records or partitions skipped.

Positioning a File by Records or Partitions

An AMP\$SKIP call positions a file by records or partitions. The valid file positioning options for a file depend on the blocking type and record type of the file:

- A file with V records and user-specified or system-specified blocking can be positioned by records or partitions.
- A file with F records and user-specified or system-specified blocking can be positioned by records, but not by partitions.
- A file with U records and user-specified blocking can be positioned by records, but not by partitions. A file with U records and system-specified blocking cannot be positioned by either records or partitions.

Specifying an invalid option on an AMP\$SKIP call returns abnormal status.

File Position After a Skip

The file position after the skip operation depends on the skip unit (records or partitions), the initial file position, the number of units skipped, and the skip direction. Table 9-1 lists the skip operation results assuming that no boundary condition is encountered before the skip count is exhausted.

NOTE

A file containing no partition delimiters is considered to contain one partition beginning at the beginning-of-information and ending at the end-of-information.

If the last record in a file is a partition delimiter and the current position of the file is after the final partition delimiter, AMP\$SKIP considers the file to be positioned at the beginning of the next partition.

Table 9-1. Skip Operation Results

File Position Before the Skip Operation	Skip Operation	Result
<i>Skipping by Records:</i>		
AMC\$BOI, AMC\$BOP, AMC\$EOR, or AMC\$EOI	Skip forward or backward zero records.	No movement; the file remains positioned the same as before the skip operation.
AMC\$MID_RECORD	Skip forward zero records.	Skips to the end of the current record.
AMC\$MID_RECORD	Skip backward zero records.	Skips to the end of the preceding record.
End of record N	Skip forward one or more (M) records.	Skips to the end of record N + M.
End of record N	Skip backward one or more (M) records.	Skips to the end of record N - M.
<i>Skipping by Partitions:</i>		
AMC\$BOI or AMC\$BOP	Skip forward or backward zero partitions.	No movement; the file remains positioned the same as before the skip operation.
AMC\$EOP, AMC\$EOR, or AMC\$MID_RECORD	Skip forward zero partitions.	Skips to the beginning of the next partition.
AMC\$, AMC\$EOR, or AMC\$MID_RECORD	Skip backward zero partitions.	Skips to the beginning of the current partition.
Beginning of partition N	Skip forward one or more (M) partitions.	Skips to the beginning of partition N + M.
Beginning of partition N	Skip backward one or more (M) partitions.	Skips to the beginning of partition N - M.
AMC\$EOP, AMC\$EOR, or AMC\$MID_RECORD	Skip forward one or more (M) partitions.	Skips to the beginning of partition N + M + 1.
AMC\$EOP, AMC\$EOR, or AMC\$MID_RECORD	Skip backward one or more (M) partitions.	Skips to the beginning of partition N - M.

Skip That Encounters File Boundaries

The information in table 9-1 assumes that no boundary conditions are encountered during the skip operation. If AMP\$SKIP encounters a boundary condition before the skip count is exhausted, it returns abnormal status. The task can determine the actual number of units skipped by calling AMP\$FETCH_ACCESS_INFORMATION to fetch the residual_skip_count value and subtracting that value from the number of units specified on the AMP\$SKIP call.

The following are the boundary conditions:

- A skip forward by records encounters a partition delimiter or the EOI.
- A skip forward by partitions encounters the EOI.
- A skip backward by records encounters a partition delimiter or the BOI.
- A skip backward by partitions encounters the BOI.

When a skip by records encounters a partition delimiter, the final position depends on whether the skip direction was forward or backward. A forward skip positions the file beyond the partition delimiter, at the beginning of the next partition. A backward skip positions the file before the partition delimiter, at the end of the preceding partition.

AMP\$REWIND

Purpose Repositions a file to its BOI.

NOTE

The access_level file attribute must be AMC\$RECORD.

Format AMP\$REWIND (file_identifier, wait, status)

Parameters **file_identifier:** amt\$file_identifier;

File identifier returned by the AMP\$OPEN call that opened the file.

wait: ost\$wait;

Procedure action after the rewind request is issued. Currently, specifying either of the following identifiers returns control to the caller when the rewind operation is complete:

OSC\$WAIT

Returns control to the caller when the operation is complete.

OSC\$NOWAIT

Currently returns control to the caller when the operation is complete. However, since this will change in a future release, it is recommended that you specify OSC\$WAIT to ensure that your program will continue to execute correctly.

status: VAR of ost\$status;

Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$conflicting_access_level
ame\$improper_file_id
ame\$ring_validation_error
ame\$unrecovered_write_error

Remarks

- If the label_type of a tape file is AMC\$UNLABELLED and the preceding operation was an output operation, the procedure terminates the volume by writing two consecutive tapemarks before rewinding the file. The file is rewound to the beginning of the first volume.
- See chapter 10 for additional information when using AMP\$REWIND to position an indexed sequential file.

AMP\$SKIP

Purpose Repositions a file forward or backward the specified number of records or partitions.

NOTE

The access_level file attribute must be AMC\$RECORD.

Format **AMP\$SKIP (file_identifier, direction, unit, count, file_position, status)**

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

direction: amt\$skip_direction;

Direction of skip.

AMC\$FORWARD

Skip forward.

AMC\$BACKWARD

Skip backward.

unit: amt\$skip_unit;

Skip unit.

AMC\$SKIP_RECORD

Skip records.

AMC\$SKIP_PARTITION

Skip partitions.

count: amt\$skip_count;

Number of units skipped (integer 0 through AMC\$FILE_BYTE_LIMIT).

file_position: VAR of amt\$file_position;

File position after skip completes.

Skip Type	Values Returned
Forward skip by records.	AMC\$EOR, AMC\$BOP, AMC\$EOI
Forward skip by partitions.	AMC\$BOP, AMC\$EOI
Backward skip by records.	AMC\$EOR, AMC\$EOP, AMC\$BOI
Backward skip by partitions.	AMC\$BOP, AMC\$BOI

status: VAR of ost\$status;

Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers

ame\$conflicting_access_level
 ame\$conflicting_fo
 ame\$improper_file_id
 ame\$improper_skip_count
 ame\$improper_skip_direction
 ame\$improper_skip_unit
 ame\$ring_validation_error
 ame\$skip_encountered_boi
 ame\$skip_encountered_bop
 ame\$skip_encountered_eoi
 ame\$skip_encountered_eop
 ame\$skip_requires_read_perm
 ame\$uncertain_tape_position
 ame\$unrecovered_write_error
 ame\$unsupported_skip

Remarks

- The procedure does not copy data to or from a user-defined working storage area or buffer space.
- Before skipping backward on an unlabeled tape file, AMP\$SKIP writes to the device any data written to the file by a previous operation.
- If the file is an unlabeled tape file and the last operation was an output operation, the procedure writes any buffered data to the tape and then terminates the volume by writing two tapemarks before skipping backward.
- See chapter 10 for additional information when using AMP\$SKIP to position an indexed sequential file.

Reading Records

To read (or get) a record means to copy data from a system buffer to a working storage area. The system reads data until it encounters a record boundary or the end of the working storage area.

File Position Returned

If the read terminates because the system encountered a record boundary, it returns the file position `AMC$EOR`. If the read terminates because the system encountered the end of the working storage area, it returns the file position `AMC$MID_RECORD`. (To read the remainder of the record, the program must issue `AMP$GET_PARTIAL` calls until the system returns the file position `AMC$EOR`.)

If the system encounters a partition boundary, it positions the file beyond the partition delimiter and returns `AMC$EOP` as the file position but transfers no data. The content of the working storage area remains the same. The next get call reads data from the first record of the next partition.

Similarly, if the system encounters the end-of-information, it returns the file position `AMC$EOI` but transfers no data. After `AMC$EOI` is returned, subsequent get calls return abnormal status (`AME$INPUT_AFTER_EOI`).

A null file is a file assigned to the null device class (the file `$NULL` is always assigned to the null device class). A get or put call to a file assigned to the null device class always returns the same file position: `AMC$EOI` for a get call, `AMC$EOR` for a full record put call, and `AMC$MID_RECORD` for a partial record put call.

Get Calls

The get calls all read data as previously described, but differ in the file position where the read begins.

`AMP$GET_DIRECT`

Reads data at the byte address specified on the call.

`AMP$GET_NEXT`

Reads data at the beginning of the next record.

`AMP$GET_PARTIAL`

Reads data either at the current file position or at the beginning of the next record.

AMP\$GET_DIRECT

Purpose Reads a record at the specified byte address.

NOTE

The `access_level` file attribute must be `AMC$RECORD`, and the `file_organization` attribute must be `AMC$BYTE_ADDRESSABLE`.

Format `AMP$GET_DIRECT (file_identifier, working_storage_area, working_storage_length, transfer_count, byte_address, file_position, status)`

Parameters

file_identifier: `amt$file_identifier`;
File identifier returned by the `AMP$OPEN` call that opened the file.

working_storage_area: `^cell`;
Working storage area address.

working_storage_length: `amt$working_storage_length`;
Number of bytes in the working storage area (integer 0 through `OSC$MAX_SEGMENT_LENGTH + 1`).

transfer_count: VAR of `amt$transfer_count`;
Number of bytes copied to the working storage area (integer 0 through `OSC$MAX_SEGMENT_LENGTH + 1`).

byte_address: `amt$file_byte_address`;
Byte address of the beginning of the record (integer 0 through `AMC$FILE_BYTE_LIMIT`).

file_position: VAR of `amt$file_position`;
File position at completion of the procedure.

`AMC$MID_RECORD`
Within a record.

`AMC$EOR`
End-of-record.

`AMC$EOP`
End-of-partition.

`AMC$EOI`
End-of-information.

status: VAR of `ost$status`;
Status record. The process identifier is `AMC$ACCESS_METHOD_ID`.

Condition	ame\$accept_bad_block
Identifiers	ame\$conflicting_access_level
	ame\$file_organization_conflict
	ame\$improper_access_attempt
	ame\$improper_file_id
	ame\$improper_input_attempt
	ame\$improper_record_address
	ame\$improper_wsl_value
	ame\$input_after_eoi
	ame\$max_cancellable_input
	ame\$ring_validation_error
	ame\$terminal_disconnected
	ame\$unrecovered_read_error

AMP\$GET_NEXT

Purpose Reads the next record.

NOTE

The access_level file attribute must be AMC\$RECORD.

Format AMP\$GET_NEXT (file_identifier, working_storage_area, working_storage_length, transfer_count, byte_address, file_position, status)

Parameters

file_identifier: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

working_storage_area: ^cell;
Working storage area address.

working_storage_length: amt\$working_storage_length;
Number of bytes in the working storage area (integer 0 through OSC\$MAX_SEGMENT_LENGTH + 1).

transfer_count: VAR of amt\$transfer_count;
Number of bytes copied to the working storage area (integer from 0 through OSC\$MAX_SEGMENT_LENGTH + 1).

byte_address: VAR of amt\$file_byte_address;
Byte address of the beginning of the record (integer 0 through AMC\$FILE_BYTE_LIMIT). The procedure returns a byte address only if the file is a mass storage file.

file_position: VAR of amt\$file_position;
File position at completion of the procedure.

AMC\$MID_RECORD

Within a record.

AMC\$EOR

End-of-record.

AMC\$EOP

End-of-partition.

AMC\$EOI

End-of-information.

status: VAR of ost\$status;
Status record. The process identifier is
AMC\$ACCESS_METHOD_ID.

**Condition
Identifiers**

ame\$accept_bad_block
ame\$conflicting_access_level
ame\$improper_access_attempt
ame\$improper_file_id
ame\$improper_input_attempt
ame\$improper_record_address
ame\$improper_wsl_value
ame\$input_after_eoi
ame\$input_after_output
ame\$max_cancellable_input
ame\$ring_validation_error
ame\$terminal_disconnected
ame\$unrecovered_read_error

For indexed sequential files only:
aae\$cant_position_beyond_bound
aae\$file_boundary_encountered
aae\$not_enough_permission
aae\$record_longer_than_wsa
aae\$wsa_not_given

Remarks

- If the current file position is not at a record boundary, the procedure repositions the file forward to the next record boundary.
- The AMP\$GET_NEXT call provides a common interface for reading records, regardless of file organization. However, when reading an indexed sequential file, an AMP\$GET_NEXT call cannot return the key separately from the record.
- When reading an indexed sequential file with embedded keys, AMP\$GET_NEXT returns the record to the working storage area. When reading an indexed sequential file with nonembedded keys, AMP\$GET_NEXT prefixes the primary key to the record and returns the key and the record together to the working storage area. Therefore, to ensure that the key and entire record can be returned, you should set the `working_storage_length` parameter to the sum of the `max_record_length` and the `key_length` attributes.
- For an indexed sequential file, an AMP\$GET_NEXT call always returns a value of zero in the `byte_address` parameter.
- See chapter 10 for additional information when using AMP\$GET_NEXT to read records from an indexed sequential file.
- When reading an indexed sequential file, AMP\$GET_NEXT returns a file position of AMC\$EOR; or, if the last record ended the list of primary keys for an alternate key value, AMC\$END_OF_KEY_LIST.

AMP\$GET_PARTIAL

Purpose Reads the specified number of bytes at the current file position.

NOTE

The `access_level` file attribute must be `AMC$RECORD`, and the `file_organization` attribute must be `AMC$SEQUENTIAL` or `AMC$BYTE_ADDRESSABLE`.

Format **AMP\$GET_PARTIAL** (`file_identifier`, `working_storage_area`, `working_storage_length`, `record_length`, `transfer_count`, `byte_address`, `file_position`, `skip_option`, `status`)

Parameters **file_identifier**: `amt$file_identifier`;
File identifier returned by the `AMP$OPEN` call that opened the file.

working_storage_area: `^cell`;
Working storage area address.

working_storage_length: `amt$working_storage_length`;
Number of bytes in the working storage area.

record_length: VAR of `amt$max_record_length`;
Number of bytes read since the previous record boundary.
If more than one call is required to read the record, the count is cumulative over the series of read operations.

transfer_count: VAR of `amt$transfer_count`;
Number of bytes copied to the working storage area.

byte_address: VAR of `amt$file_byte_address`;
Byte address of the beginning of the record. The procedure returns a byte address only if the file is a mass storage file.

file_position: VAR of amt\$file_position;
File position at completion of the procedure.

AMC\$MID_RECORD

Within a record.

AMC\$EOR

End-of-record.

AMC\$EOP

End-of-partition.

AMC\$EOI

End-of-information.

skip_option: amt\$skip_option;

Indicates whether the procedure repositions the file before reading data.

AMC\$SKIP_TO_EOR

If the current file position is not at the beginning of a record, the procedure repositions the file forward to the next record boundary.

AMC\$NO_SKIP

The procedure does not reposition the file.

status: VAR of ost\$status;

Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

**Condition
Identifiers**

ame\$accept_bad_block
ame\$conflicting_access_level
ame\$improper_access_attempt
ame\$improper_file_id
ame\$improper_input_attempt
ame\$improper_skip_option
ame\$improper_wsl_value
ame\$input_after_eoi
ame\$input_after_output
ame\$max_cancellable_input
ame\$ring_validation_error
ame\$terminal_disconnected
ame\$unrecovered_read_error

AMP\$GET_PARTIAL

Remarks If the skip_option parameter specifies AMC\$SKIP_TO_EOR and the current file position is not at the beginning of a record, the procedure repositions the file forward to the next record boundary.

Writing Records

To put a record means to copy data from a working storage area to a system buffer. The system performs the physical I/O operations required to write data from the buffer to the file.

The `working_storage_length` parameter on the `put` call specifies the number of bytes copied (starting at the address specified as the `working_storage_address`). An `AMP$PUT_DIRECT` or `AMP$PUT_NEXT` call puts an entire record; an `AMP$PUT_PARTIAL` call can put part of a record, allowing you to write records longer than your working storage area.

The file location at which the write begins depends on the `put` call used.

`AMP$PUT_DIRECT`

Writes data at the specified byte address.

`AMP$PUT_NEXT`

Writes data at the current file position.

`AMP$PUT_PARTIAL`

Writes data at the current file position.

Establishing a New End-of-Information

A `put` call establishes a new end-of-information for a sequential file. For `AMP$PUT_NEXT` and `AMP$PUT_PARTIAL`, the new end-of-information is always immediately after the data just written. Data that was previously written after that address is no longer accessible.

For `AMP$PUT_DIRECT`, a new end-of-information is established only if the writing of the record lengthens the file. It determines this by checking whether the working storage length added to the specified byte address exceeds the current end-of-information address.

Writing an end-of-partition at the file `EOI` extends the file to include the end-of-partition indicator (a zero-length `V` record).

Padding Fixed-Length Records

For `AMP$PUT_DIRECT` and `AMP$PUT_NEXT`, if the record type of the file is `AMC$ANSI_FIXED` and the working storage area length is shorter than the fixed record length, the system pads the record. To do so, it appends padding characters until the record is the fixed record length. `AMP$PUT_PARTIAL` also pads `AMC$ANSI_FIXED` records, but only when the call writes the last part of the record.

Truncating Fixed-Length Records

When making AMP\$PUT_DIRECT and AMP\$PUT_NEXT calls, and the record type is AMC\$ANSI_FIXED, if the working storage length exceeds the fixed record length, the record is truncated to the fixed record length as it is written.

Writing Records Longer Than the Working Storage Area

Successive AMP\$PUT_PARTIAL calls can write a record whose cumulative length exceeds the working storage area length. The call specifies whether the data is to be written as the first part of the record, a middle part of the record, or the last part of the record.

At any time, an AMP\$PUT_PARTIAL call can specify that the data is the starting part of a record. If the file position is AMC\$MID_RECORD, the call terminates the current record before writing the data as the beginning of a new record. The file position after the call is always AMC\$MID_RECORD.

If the AMP\$PUT_PARTIAL call specifies that the data is a middle part of a record, it writes the data at the current file position. The file position before and after the call is always AMC\$MID_RECORD.

If the AMP\$PUT_PARTIAL call specifies that the data is the ending part of the record, the procedure writes the data and then terminates the record. The file position after the call is always AMC\$EOR. If the file position before the call is AMC\$EOR, the action taken is the same as that of an AMP\$PUT_NEXT call.

AMP\$FLUSH

Purpose Writes all modified file data in memory to the device to which the file is assigned.

NOTE

This procedure is valid only for files assigned to interactive terminals and files with indexed sequential file organization.

Format AMP\$FLUSH (**file_identifier**, **wait**, **status**)

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

wait: ost\$wait;

Action to be taken after the flushing operation is initiated.

NOTE

The wait parameter is ineffective for interactive terminal files because the flush operation is always completed before control is returned to the user (OSC\$WAIT option).

OSC\$WAIT

Complete the flush operation before returning control to the caller.

OSC\$NOWAIT

Return control to the caller immediately. To determine whether the flush operation has completed, the program must call AMP\$FETCH_ACCESS_INFORMATION and check the value of the last_op_status item.

status: VAR of ost\$status;

Status record. The process identifier is AMC or AAC\$ACCESS_METHOD_ID.

AMP\$FLUSH

Condition Identifiers	<code>ame\$improper_file_id</code> <code>ame\$ring_validation_error</code> For indexed sequential files only: <code>aae\$bad_block_table_overflow</code> <code>aae\$file_reached_file_limit</code> <code>aae\$no_updates_till_recovered</code> <code>aae\$not_enough_permission</code> <code>aae\$system_routine_failed</code> <code>aae\$write_parity_error</code>
Remarks	<ul style="list-style-type: none">• Flushing data to an indexed sequential file updates the data on mass storage to reflect the current data in memory. This ensures that if the data in memory is lost due to a system failure, the updated data is still available in the mass storage file.• Flushing data destined for an interactive terminal ensures that all output has been sent to the terminal. Interactive terminal output is buffered if NAM cannot output data as quickly as the program sends it. A task could call AMP\$FLUSH to ensure that all output has been displayed at the terminal before it continues processing.• A get or close call for a terminal file flushes all output data to the terminal before performing the get or close operation.• A close call for a permanent mass storage file flushes all output data to mass storage before the call terminates. If the data cannot be successfully written, AMP\$CLOSE returns abnormal status. AMP\$CLOSE closes the instance of open regardless of any I/O errors.

AMP\$PUT_DIRECT

Purpose Writes a record at the specified byte address.

NOTE

The `access_level` file attribute must be `AMC$RECORD`, and the `file_organization` attribute must be `AMC$BYTE_ADDRESSABLE`.

Format `AMP$PUT_DIRECT (file_identifier, working_storage_area, working_storage_length, byte_address, status)`

Parameters

file_identifier: `amt$file_identifier`;
File identifier returned by the `AMP$OPEN` call that opened the file.

working_storage_area: `^cell`;
Working storage area address.

working_storage_length: `amt$working_storage_length`;
Number of bytes in the working storage area (0 through `OSC$MAX_SEGMENT_LENGTH + 1`).

byte_address: `var of amt$file_byte_address`;
Byte address of the beginning of the record (0 through `AMC$FILE_BYTE_LIMIT`).

status: `VAR of ost$status`;
Status record. The process identifier is `AAC$ACCESS_METHOD_ID` or `AMC$ACCESS_METHOD_ID`.

- Remarks**
- If the current file position is AMC\$MID_RECORD, the procedure terminates the current record before writing the next record.
 - If the file is a mass storage file, the procedure returns the starting byte address of the record. If the task stores the starting byte address of the record, the record could later be accessed by address using an AMP\$GET_DIRECT call.
 - If the record type is F and the working storage length is less than the max_record_length, the record is padded to the fixed record length using the padding character. When the working storage length is greater than the max_record_length, the record is truncated to the fixed_record_length.
 - AMP\$PUT_NEXT can write records sequentially to any file regardless of its file organization. When writing to an indexed sequential file with embedded keys, AMP\$PUT_NEXT assumes that the record key is embedded in the record data within the working storage area as indicated by the key_position attribute value. When writing to an indexed sequential file with nonembedded keys, AMP\$PUT_NEXT assumes that the key is at the beginning of the working storage area. It removes the number of bytes specified by the key_length attribute from the beginning of the working storage area and writes those bytes as the nonembedded key. It then writes the bytes remaining in the working storage area as the data record. When AMP\$PUT_NEXT is used for indexed sequential files, a value of zero is always returned in the byte_address parameter.

AMP\$PUT_PARTIAL

Purpose Writes a partial record at the current byte address.

NOTE

The `access_level` file attribute must be `AMC$RECORD`, and the `file_organization` attribute must be `AMC$SEQUENTIAL` or `AMC$BYTE_ADDRESSABLE`.

Format `AMP$GET_PARTIAL (file_identifier, working_storage_area, working_storage_length, byte_address, term_option, status)`

Parameters `file_identifier`: `amt$file_identifier`;
File identifier returned by the `AMP$OPEN` call that opened the file.

`working_storage_area`: `^cell`;
Working storage area address.

`working_storage_length`: `amt$working_storage_length`;
Number of bytes in the working storage area (integer from 0 through `OSC$MAX_SEGMENT_LENGTH + 1`).

`byte_address`: `VAR` of `amt$file_byte_address`;
Byte address of the beginning of the record (integer 0 through `AMC$FILE_BYTE_LIMIT`). The procedure returns a value only if the file is a mass storage file.

`term_option`: `amt$term_option`;
Record part to be written.

`AMC$START`
First part of the record.

`AMC$CONTINUE`
Middle part of the record.

`AMC$TERMINATE`
Last part of the record.

`status`: `VAR` of `ost$status`;
Status record. The process identifier is `AMC$ACCESS_METHOD_ID`.

Condition ame\$conflicting_access_level
Identifiers ame\$improper_access_attempt
 ame\$improper_continue
 ame\$improper_file_id
 ame\$improper_output_attempt
 ame\$improper_term_option
 ame\$improper_wsl_value
 ame\$put_beyond_file_limit
 ame\$record_exceeds_mbl
 ame\$ring_validation_error
 ame\$terminal_disconnected
 ame\$unrecovered_write_error

Remarks: If the file is a mass storage file, the procedure returns the starting byte address of the record. (The same address is returned by each call that writes part of the record.) If the task stores the starting byte address of the record, the record could later be accessed by address using an AMP\$GET_DIRECT call.

Writing Partition Delimiters

Writing a partition delimiter groups preceding records into a partition. If the delimiter is the first partition delimiter on the file, the partition comprises the records between the beginning of the file and the partition delimiter; otherwise, the partition comprises the records between the previous partition delimiter and the current partition delimiter.

Of the supported record types, only the V record type supports partitions. Therefore, the AMP\$WRITE_END_PARTITION call that writes a partition delimiter is effective only when the file record type is AMC\$VARIABLE, and the file organization is sequential or byte-addressable.

AMP\$WRITE_END_PARTITION

Purpose Writes a partition delimiter at the current file position.

NOTE

The `access_level` file attribute must be `AMC$RECORD`, and the `record_type` attribute must be `AMC$VARIABLE`.

Format **AMP\$WRITE_END_PARTITION (file_identifier, status)**

Parameters **file_identifier**: amt\$file_identifier;
File identifier returned by the AMP\$OPEN call that opened the file.

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$conflicting_access_level
ame\$improper_file_id
ame\$improper_output_attempt
ame\$partitioning_unsupported
ame\$ring_validation_error



Using the Indexed Sequential File Organization

Primary Keys	10-1
Indexed Sequential File Structure	10-3
Data Blocks	10-3
Index Blocks	10-6
Creating an Indexed Sequential File	10-10
Setting File Attributes	10-10
Writing Records	10-15
AMP\$PUT_KEY	10-16
Processing an Existing Indexed Sequential File	10-18
Setting Temporary Attribute Values	10-19
Positioning the File	10-19
AMP\$START	10-21
Reading Records	10-24
AMP\$GET_KEY	10-25
AMP\$GET_NEXT_KEY	10-28
Replacing and Deleting Records	10-30
AMP\$PUTREP	10-31
AMP\$REPLACE_KEY	10-33
AMP\$DELETE_KEY	10-35
Monitoring the Index Levels in an Indexed Sequential File	10-37
Recreating an Indexed Sequential File	10-37
Indexed Sequential File Example	10-39
Alternate Keys	10-47
The Alternate Index	10-47
Alternate Key Definition	10-48
Creating and Deleting Alternate Keys	10-57
AMP\$CREATE_KEY_DEFINITION	10-58
AMP\$DELETE_KEY_DEFINITION	10-64
AMP\$APPLY_KEY_DEFINITIONS	10-65
AMP\$ABANDON_KEY_DEFINITIONS	10-67
Using Alternate Keys	10-68
AMP\$SELECT_KEY	10-74
AMP\$GET_KEY_DEFINITIONS	10-75
AMP\$GET_PRIMARY_KEY_COUNT	10-78
AMP\$GET_NEXT_PRIMARY_KEY_LIST	10-81
Alternate Key Example	10-84



Using the Indexed Sequential File Organization

10

Besides the sequential and byte-addressable file organizations, NOS/VE also supports the indexed sequential file organization. The indexed sequential file organization allows direct access to each record in the file through the unique key value associated with each record.

Primary Keys

Within an indexed sequential file, data is stored as records. Each data record is associated with a unique value called its primary key. A data record is associated with its primary key when the record is written to the file.

The primary key can be embedded in the data (an embedded key) or separate from the data (a nonembedded key). Each primary key value is unique within the file; there can be no duplicate primary key values in a file.

When an indexed sequential file is read sequentially, its records are accessed in order by ascending key value. This sorted order is kept even when new records are added to the file.

Part 1 of figure 10-1 shows an unsorted sequence of records. Assume these records are written to an indexed sequential file with the first field, the employee number, specified as the primary key. If the records were then read sequentially from the file, they would be read in sorted order by employee number. Part 2 of figure 10-1 shows the records sorted by employee number.

In the figure 10-1 example, the primary key is the employee number because each employee has a unique number. The employee's last name could not be used as the primary key because two employees have the same last name.

1. Unsorted Records			
39248	Miller	Robert	Driver
42976	Stevens	Carol	Manager
39048	Jetson	Harry	Asst Manager
51234	Miller	Catherine	Secretary
82176	Beirmeyer	William	Driver
75090	Arnold	Terry	Computer Operator
49257	Lane	Gladys	Accountant
38602	Johnstone	Mark	Computer Operator
13905	McGuire	Stewart	Clerk
2. Sorted Records			
13905	McGuire	Stewart	Clerk
38602	Johnstone	Mark	Computer Operator
39048	Jetson	Harry	Asst Manager
39248	Miller	Robert	Driver
42976	Stevens	Carol	Manager
49257	Lane	Gladys	Accountant
51234	Miller	Catherine	Secretary
75090	Arnold	Terry	Computer Operator
82176	Beirmeyer	William	Driver

Figure 10-1. Records Sorted by Primary Key

Indexed Sequential File Structure

Unlike the sequential and byte addressable file organizations, the structure of an indexed sequential file has a more than one component. The components of the file structure are the internal file label, data blocks and index blocks.

You cannot access the internal file label or directly change information contained in the label; the internal file label is for system use only.

Data Blocks

Records in an indexed sequential file are grouped into data blocks. All data blocks in the file are the same size. Each block contains a header, data records, padding, and record pointers.

Figure 10-2 shows the structure of a data block that contains four records. The block header is at the beginning of the block and is immediately followed by four data records. The four record pointers are at the end of the block. The empty space between the last data record and the record pointers is the data block padding that allows for insertion of additional records into the data block.

Block Header	
Record 1	
Record 2	
Record 3	
Record 4	
Empty Space	
Record Pointer 4	Record Pointer 3
Record Pointer 2	Record Pointer 1

Figure 10-2. Indexed Sequential File Data Block Structure

Data Block Record Pointers

The record pointers in each data block provide direct access to each record in the block. If all records in a block are the same length, only one record pointer is needed; otherwise, one pointer per record is needed.

Record pointers are stored at the end of the data block, beginning with the last byte. Each pointer requires three bytes of storage. This means that the record pointer for the first record in the data block is stored in the last three bytes in the block, the record pointer for the second record is stored in the next to the last three bytes, and so forth (see figure 10-2).

Data Block Padding

When an indexed sequential file is created, each data block can be created with extra space for later insertion of records. The extra empty space is called data block padding.

The amount of data block padding is a percentage of the space in each block. For example, if the data block padding is 25 percent, records and record pointers are written to the block until it is 75 percent full. The system then stops writing records in that block and starts a new block.

The data block padding percentage is a file attribute value. However, the attribute value is used only when the file is created. It is not used after the initial instance of open of the file.

If you are certain that an indexed sequential file will only be read and never written after file creation, you should specify 0 as the data block padding percentage. In this case, no extra space is left for later record insertion. It does not mean that records cannot be inserted; it just means that an insertion would result in an immediate data block split.

Data Block Split

When a data record is added to an indexed sequential file, it is stored so as to maintain the sorted order of the data records. For example, if a record with primary key value 3 is added, it must be stored between the records with primary key values 2 and 4.

If the data block in which the record should be inserted does not have enough empty space for the record, a data block split occurs. Records in the data block which precede the new record remain in the existing block. All records in the existing block that come after the new record are moved to the newly created block. The new record is put into either the new block or the existing block depending on the amount of empty space in the blocks and the size of the new record. If the new record does not fit in either block, a second new block is created and the new record is put into this block.

Figure 10-3 shows an example of a data block split. Part I shows data block A before record 3 is inserted. To keep the indexed sequential file in order, record 3 must be inserted between record 2 and record 4; therefore, a data block split occurs. Part II shows the result of the block split. Record 1 and record 2 remain in data block A, while record 4 and record 5 are moved to data block B. Record 3 is inserted in data block A because data block A has more empty space than data block B.

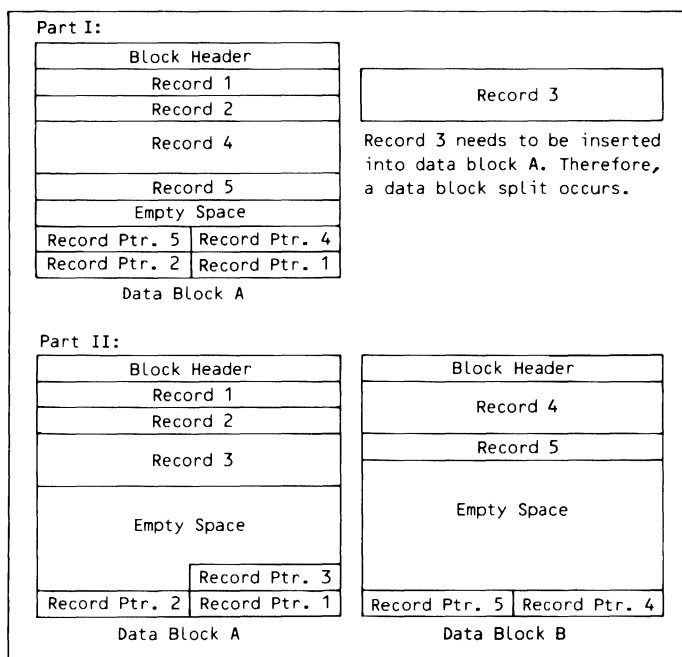


Figure 10-3. Data Block Split

Index Blocks

To access a data record, the system must know which data block contains the record. To do so, it searches the index blocks.

Index blocks are the same size as data blocks. Like data blocks, each index block contains a header, a sequence of records, empty space, and a record pointer.

The structure of an index block is illustrated in figure 10-4.

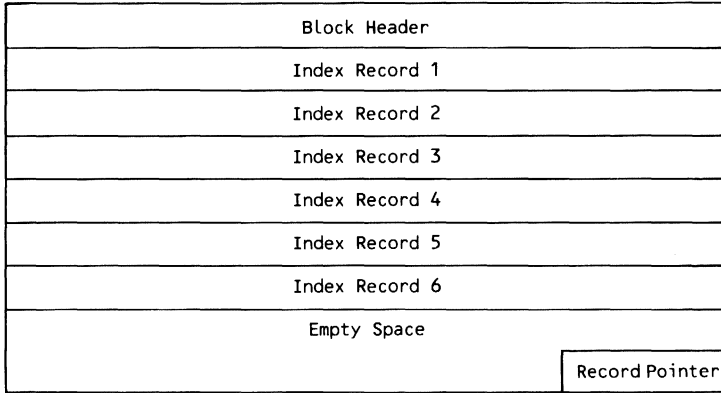


Figure 10-4. Indexed Sequential File Index Block Structure

Index Records

Each index record contains the value of the primary key for the first record in another block and the physical address of the block. The block to which the index record points can be either a data block or a lower-level index block.

Index records are stored in ascending order according to primary key value in each record. An index record is used to access all records with primary key values between its key and the key in the next index record.

Index Block Record Pointer

Each index block contains only one record pointer because all index records are the same length and so only one pointer is needed to access each record in the block. The pointer is always the last three bytes in the index block.

Index Block Padding

Like data blocks, a percentage of the space in each index block is left empty when the file is created. The space is used for the insertion of new index records. New index records are inserted when new data blocks are created due to a data block split.

The index block padding percentage is a file attribute value. However, the attribute value is used only when the file is created. It is not used after the initial instance of open of the file.

Index Levels

All indexed sequential files, except those consisting of a single data block, have at least one index block. If the file has more than one index block, the blocks are linked in a hierarchy. The topmost block in the hierarchy, the level 0 block, contains an index record for each index block on the next lower level, the level 1 blocks. The level 1 blocks contains index records for each index block at level 2, and so forth.

A maximum of 15 levels of index blocks (numbered 0 to 14) is allowed; however, performance is usually best when no more than two index levels exist. New index levels are created due to index block splits.

Index Block Split

When the first data record is written to the file, no index blocks exist. When the first data block is full and a second data block is needed, the system creates a primary (level 0) index block. The system stores an index record in the level 0 index block for each new data block it creates.

Index block splits create the hierarchy of index blocks. When the level 0 index block cannot hold any more index records, the system creates a second index block. The new index block and the existing index block become level 1 index blocks and contain index records for the data blocks. At the same time, a new level 0 index block is created to hold index records referencing the level 1 index blocks. Additional level 1 index blocks are added until the primary index block is filled and another level of indexing is needed.

For example, figure 10-5 illustrates a search through two index levels. Suppose you wanted to access the record with primary key 43. The system always begins its search with the level 0 index block. It searches the level 0 block and finds an index record for primary key 15 followed by an index record for primary key 100. It then follows the pointer in the primary key 15 index record to a level 1 index block. There, it finds an index record for primary key 25 followed by a key for primary key 55. It then follows the pointer for the index record for primary key 25 to a data block. It then searches the data block for the data record having primary key 43.

The percentage of empty space left in each index block when it is created (that is, the index block padding) determines how much space is left for insertion of additional index records and thus, when an index block split is required. The index block padding percentage is specified by the index_ padding attribute value.

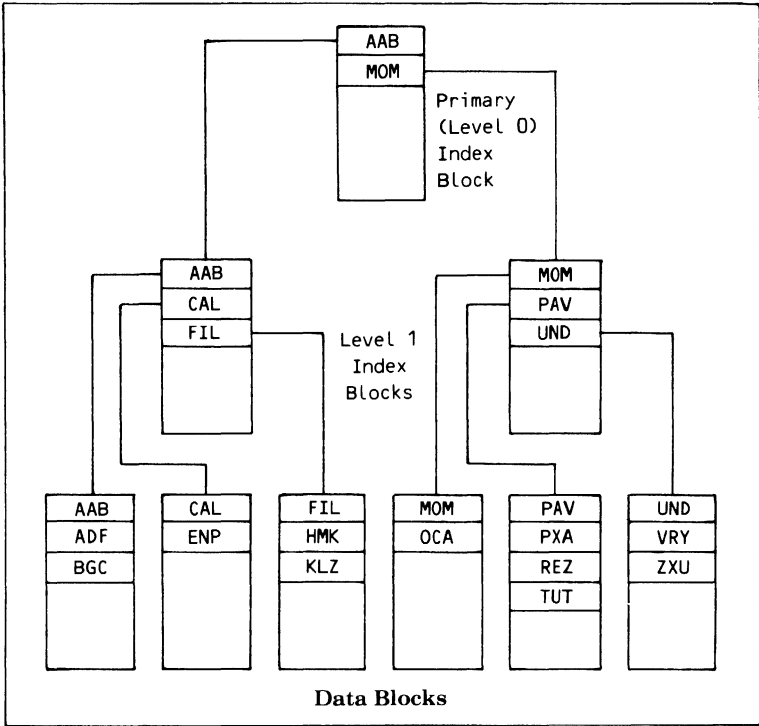


Figure 10-5. Record Search Through Two Index Levels

Creating an Indexed Sequential File

To create an indexed sequential file, a task performs the following steps:

1. Set file attributes (AMP\$FILE or AMP\$OPEN calls).
2. Open the file (AMP\$OPEN call).
3. Write records to the file (AMP\$PUT_KEY or AMP\$PUT_NEXT calls).
4. Close the file (AMP\$CLOSE call).

As described here, the file data is written in the same instance of open that created the file. However, this is not required; the file data can be written during a later instance of open.

Setting File Attributes

Before opening an indexed sequential file for the first time, you must set file attributes defining the structure of the file and processing limitations for the file. When the new file is opened, the file attributes are stored in the internal label of the file, and the system references the attribute values whenever the file is processed.

As described in chapter 6, *Defining File Attributes*, the attributes that define the file structure cannot be changed after the file is first opened. Chapter 6 describes the means of setting file attribute values.

You should select file attribute values carefully. Selecting suitable values for file attributes helps ensure that the file economizes space and the time needed for record retrievals.

Required File Attributes

The file attribute listing in chapter 6, *Defining File Attributes*, describes all file attributes. As indicated in the attribute description, certain file attributes are effective for indexed sequential files only. Other file attributes are effective for all file organizations, but have additional processing rules for indexed sequential files. Therefore, to ensure that attribute values are specified correctly for indexed sequential files, you should read the attribute description before defining an attribute value.

For an indexed sequential file, you must define values for the following attributes:

- `file_organization`: Must be set to `AMC$INDEXED_SEQUENTIAL`.
- `key_length`: No default value provided; `AMP$OPEN` returns a fatal error if undefined.
- `max_record_length`: No default value provided; `AMP$OPEN` returns a fatal error if undefined.

These attributes and the attributes described in the following paragraphs are preserved with the file and cannot be changed after the file is first opened.

Defining the Record Type and Length

You must establish the record type, minimum record length, and maximum record length before the new file is first opened. These are defined by the `record_type`, `min_record_length`, and `max_record_length` attributes.

The valid values for the `record_type` attribute are `AMC$VARIABLE`, `AMC$ANSI_FIXED`, or `AMC$UNDEFINED`; the default record type for indexed sequential files is `AMC$UNDEFINED`.

You must specify a value for the `max_record_length` attribute; it has no default value. The default value for the `min_record_length` attribute depends on the values of the `embedded_key` and `record_type` attributes:

- If the `record_type` is `AMC$ANSI_FIXED`, the default `min_record_length` value is the `max_record_length` value.
- If the `record_type` is `AMC$VARIABLE` or `AMC$UNDEFINED` and the `embedded_key` attribute is `TRUE`, the default `min_record_length` value is the sum of the `key_position` and `key_length` values.
- If the `record_type` is `AMC$VARIABLE` or `AMC$UNDEFINED` and the `embedded_key` attribute is `FALSE`, the default `min_record_length` value is 1 byte.

The `min_record_length` value cannot exceed the `max_record_length` value. If the `record_type` is `AMC$ANSI_FIXED`, the `min_record_length` value must be the same as the `max_record_length` value.

If the primary key is embedded in the record, the primary key field must be within the minimum record length. Therefore, the `key_length` value cannot exceed the `min_record_length` value. An attempt to write a record smaller than the minimum record length or longer than the maximum record length is rejected with a trivial error.

Defining the Primary Key

In an indexed sequential file, each data record must have a unique primary key value. Before opening a new indexed sequential file, you must define the primary key by setting or accepting defaults for the `key_type`, `embedded_key`, and `key_position` attributes.

Setting Key_Type

The `key_type` attribute defines the primary key type for the file and can be set as follows:

AMC\$UNCOLLATED_KEY

Keys (1 through 255 bytes) ordered byte-by-byte according to the ASCII character set sequence (listed in appendix B). The key can be a positive integer or a string of ASCII character codes.

AMC\$INTEGER_KEY

Integer keys (1 through 8 bytes) ordered numerically. The integer can be positive or negative.

AMC\$COLLATED_KEY

Collated keys are 1- through 255-character keys ordered according to the collation table you specify as the `collate_table_name` attribute. If you specify this key type, you must supply a collation table; there is no system-supplied default collation table.

Appendix E lists the predefined collation tables. Primary keys are not stored in collated form. (The system uses hardware instructions for collated key operations.) Therefore, the collation tables can map more than one character to the same position in the collating sequence. For example, several predefined tables map the 256 ASCII characters to 64 collating positions.

If you do not specify a value for the `key_type` attribute, the value `AMC$UNCOLLATED_KEY` is used.

Setting Embedded Key and Key Position

The `embedded_key` attribute determines whether the primary key for a record is located in the record (embedded) or is separate from the record (nonembedded).

If the value of the `embedded_key` attribute is `TRUE`, the primary keys are embedded. If the attribute value is `FALSE`, the keys are nonembedded. If you do not set a value for the `embedded_key` attribute, the value `TRUE` is used.

For files with embedded keys, you must also set the `key_position` attribute or accept the default value of zero. The system uses the `key_position` attribute to locate the first byte of the primary key.

Defining the Block Size

Data blocks and index blocks are the same size. You can specify the block size explicitly using the `max_block_length` attribute or accept the default block size calculated by the system.

It is recommended that you allow the system to calculate block size.

You specify block size explicitly by setting the `max_block_length` attribute. If you specify a value for `max_block_length`, the system increases the value, if necessary, so that a block can hold at least one maximum-length record. Then the value is rounded up to the next power of 2 between 2,048 and 65,536 bytes, inclusive.

If you do not specify a value for the `max_block_length` attribute, the system uses the values of the following attributes to calculate block size:

```
average_record_length
estimated_record_count
index_levels
max_record_length
min_record_length
records_per_block
```

If you decide to let the system calculate block size, you should set as many of these attributes as possible.

The system calculates block size as follows:

1. The value of the `average_record_length` attribute is used as the average length of the records in the file. If you did not specify a value for `average_record_length`, the arithmetic mean of the `max_record_length` and `min_record_length` attributes is used as the average record length; however, the system does not store this value as the `average_record_length` attribute.

When the indexed sequential file has `AMC$VARIABLE` or `AMC$UNDEFINED` type records, you should determine the value of the `average_record_length` attribute as follows:

- If most records in the file are of a specific length, the value of `average_record_length` should be set to that length.
- If record lengths are well distributed, the value of `average_record_length` should be set to the median of the record lengths; that is, half the records are smaller and half are larger than the value of `average_record_length`.

If keys are nonembedded, the value of `average_record_length` should be determined without regard to the value of the `key_length` attribute.

2. The value of the `records_per_block` attribute is used as an estimate of the number of records a data block should contain. If `records_per_block` is not specified, the estimate of two records per block is used.

The value of the `records_per_block` attribute has only a small effect on the calculation of block size and is used only for the calculation; it is not a limit to the number of records a block can contain.

3. The value of the `estimated_record_count` attribute is used as the estimated maximum number of records in the file. If `estimated_record_count` is not specified, the value of `record_limit` is used. If `record_limit` is not specified, the estimate of 100,000 records is used.
4. The value of the `index_level` attribute is used as the estimate of the number of index levels for the file. If `index_level` is not specified, the estimate of two index levels is used.
5. The system determines the smallest block size so that it can contain both:
 - The number of records specified by the `records_per_block` attribute with each record the length specified by the `average_record_length` attribute.
 - Sufficient index records so that if the file grows to its estimated maximum number of data records (`estimated_record_count` value), the number of index levels will be within the maximum (`index_level` value).

A large block size is efficient in that it minimizes the number of index levels. However, in extreme cases, a large block size can reduce efficiency because a larger block will have to be read to memory each time a record is randomly accessed. For accessing records sequentially, a larger data block is always more efficient because fewer blocks will be read to memory.

Defining Data_Padding and Index_Padding

The `data_padding` and `index_padding` attributes specify the percentage of data block and index block padding, respectively (0 through 99). The default value for both attributes is zero, no padding.

Data block padding should be used only if the records being inserted during the creation of the file have been already sorted by primary key. In this case, data block padding helps avoid data block splits when the file is updated. Also, data block padding should be used with care because the padding is allocated in every data block in the data file.

Although index blocks are the same size as data blocks, the percentage of index block padding need not be the same as for data block padding. A small percentage of index block padding is usually recommended if a number of updates to the file is expected. If you specify index block padding at file creation time, index records can be added without creating additional index levels.

Writing Records

After the indexed sequential file is opened, records can be written to the file using the `AMP$PUT_KEY` or the `AMP$PUT_NEXT` calls.

A write operation copies the data moved from the working storage area to the file. The value of the primary key determines the logical position of the record in the file.

Whenever you write a large number of records to the file, such as is usually done when the file is created, you should presort the records in ascending order by primary key value. Presorting records can result in a smaller file and less time required for writing the records. You can sort records using `NOS/VE Sort/Merge` as described in the `Sort/Merge` manual.

You can write records sequentially using either `AMP$PUT_KEY` or `AMP$PUT_NEXT` calls. Use of `AMP$PUT_KEY` calls is recommended for writing indexed sequential files. `AMP$PUT_NEXT` should be used only if a common interface for writing records, regardless of file organization, is required.

AMP\$PUT_KEY

Purpose Writes a record to an indexed sequential file.

NOTE

The file must be open with at least PFC\$APPEND permission.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$PUT_KEY (**file_identifier**, **working_storage_area**, **working_storage_length**, **key_location**, **wait**, **status**);

Parameters **file_identifier**: amt\$file_identifier
File identifier returned by AMP\$OPEN for the file.

working_storage_area: ^cell
Pointer to the new record.

working_storage_length: amt\$working_storage_length
Length, in bytes, of the record to be written.

key_location: ^cell
Pointer to the primary key of the new record; specify NIL if the primary keys are embedded.

wait: ost\$wait
Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR ost\$status
Status variable in which the completion status is returned.

Condition Identifiers

aae\$duplicate_alternate_key
 aae\$file_at_file_limit
 aae\$file_at_user_record_limit
 aae\$file_full_no_puts_or_reps
 aae\$file_is_ruined
 aae\$key_already_exists
 aae\$key_required
 aae\$nonembedded_key_not_given
 aae\$not_enough_permission

Remarks

- If the primary key is nonembedded, the `key_location` parameter specifies the starting address of the key. If the primary key is embedded, the `key_location` parameter is ignored, and the location of the key is determined by the `key_position` attribute; therefore, the `key_location` parameter should be specified as NIL.
- If the file has `AMC$ANSI_FIXED` records, the `working_storage_length` parameter is ignored, and the value of the `max_record_length` attribute is used as the length of the working storage area.
- Writing records to an indexed sequential file is usually faster if the records are sorted in ascending primary key order before being written to the file. Also, the resulting file will probably be smaller.
- Writing unsorted records to an indexed sequential file could result in inefficient file structure with more data blocks than necessary because of numerous data block splits. Also, more time is required to create the file.
- An `AMP$PUT_KEY` call updates the alternate indexes for the new record if alternate keys are defined for the file. Calls to `put` or `replace` records are effective even if an alternate key is currently selected for reading and positioning the file.

Processing an Existing Indexed Sequential File

After you create an indexed sequential file, you can perform these functions on the file:

- Read records.
- Write records.
- Delete records.
- Replace existing records.
- Define alternate keys as described later in this chapter.

File processing is governed by the file attributes specified when the file was created.

To use or change the data in an existing indexed sequential file, a task performs the following steps:

1. Sets temporary file attribute values, if desired (AMP\$FILE or AMP\$OPEN calls).
2. Opens the file (AMP\$OPEN call).
3. Positions the file to a specific record, if desired (AMP\$GET_KEY, AMP\$REWIND, AMP\$SKIP or AMP\$START calls).
4. Accesses records in the file:
 - Reads records (AMP\$GET_KEY, AMP\$GET_NEXT_KEY, or AMP\$GET_NEXT calls).
 - Writes records (AMP\$PUT_KEY, AMP\$PUT_NEXT, or AMP\$PUTREP calls).
 - Replaces records (AMP\$REPLACE_KEY or AMP\$PUTREP calls).
 - Deletes records (AMP\$DELETE_KEY calls).
5. Closes the file (AMP\$CLOSE call).

Depending on the value of the forced_write attribute, the system might not write modified blocks to mass storage immediately after the modification. AMP\$FLUSH can be used any time after the file is opened to write all blocks to mass storage. Execution of the AMP\$FLUSH call does not change the position of the file.

Setting Temporary Attribute Values

You can change the values of temporary attributes before or after the file is opened. The attribute values you can specify and the means of changing old file attributes are described in chapter 6, *Defining File Attributes*.

The following temporary attributes are effective only for indexed sequential files:

- `error_limit`: Sets a limit on the number of trivial errors that can occur. A trivial error is an error that prevents successful completion of the current request, but does not prevent processing of subsequent requests. The system declares a fatal error when the trivial error limit is reached.
- `message_control`: Determines the types of information written on the `$ERRORS` file (trivial errors, informative messages, and/or statistics).

Positioning the File

A task can use `AMP$GET_KEY`, `AMP$REWIND`, `AMP$SKIP`, and `AMP$START` calls to position an indexed sequential file. The calls position the file as follows:

- `AMP$GET_KEY`: Returns to the working storage area the record whose key value matches the key specified on the call and positions the file at the end of the returned record.
- `AMP$REWIND`: Positions a file in front of the record with the lowest key value.
- `AMP$SKIP`: Positions a file forward or backward one or more records.
- `AMP$START`: Positions a file to the beginning of the record whose key value matches the key specified on the call.

To use a positioning call on a file, you must open the file for record access with at least read access permission.

For information on a positioning a file by alternate key values, refer to *Using Alternate Keys* later in this chapter.

Positioning a File by Major Key

The AMP\$START and AMP\$GET_KEY calls have a major_key_length parameter. This parameter allows a call to position the file according to a major key value.

A major key consists of one or more of the leftmost bytes of a key. The major_key_length parameter specifies the number of bytes to use as the major key. A major key search compares only the number of bytes in the major key.

For example, suppose the key at the specified key_location is ABCDEF and the major_key_length parameter value is 2. The major key is the leftmost two bytes, characters AB. The major key search compares the characters AB with the leftmost two bytes of the searched keys. It positions the file at the first record whose key begins with AB or greater.

As a second example, suppose the key specified on the call is the hexadecimal integer FF145 and the major key length value is 3. The major key used is the leftmost three bytes containing the value FF1 so the file is positioned at the first record whose key begins with FF1 or greater.

If the major_key_length parameter is zero or equal to key_length, the entire key is used to position the file.

AMP\$START

Purpose Positions the file to the beginning of the first record in the file having a key that satisfies the specified key relation. A record is not returned to the working storage area.

NOTE

The file must be open with at least PFC\$READ permission.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$START (file_identifier, key_location, major_key_length, key_relation, file_position, wait, status);

Parameters **file_identifier:** amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

key_location: ^cell

Location of the key to which the key of each record in the file is compared.

major_key_length: amt\$major_key_length

Length of the major key in bytes. The major key is the leftmost bytes of the key at key_location. The major key is compared to the leftmost bytes of a key.

If the value is zero, a full-length key is used to position the file. Otherwise, the number of bytes specified for the major_key_length parameter must be less than or equal to the value of the key_length attribute.

key_relation: amt\$key_relation

Relationship between the key of the record and the key at key_location. The possible values are as follows:

AMC\$EQUAL_KEY

The key of the record must be equal to the key at key_location.

AMC\$GREATER_OR_EQUAL_KEY

The key of the record must be equal to the key at key_location or, if an equal key does not exist, must be the next greater key value.

AMC\$GREATER_KEY

The key of the record must be the first key value greater than the key at key_location.

file_position: VAR amt\$file_position

File position at completion of the start operation.

AMC\$END_OF_KEY_LIST

File is positioned to read the first record containing the alternate key value specified on the call (that is, at the end of the preceding key list, if one exists).

AMC\$EOR

File is positioned to access the record containing the primary key value specified on the call (that is, at the end of the preceding record, if one exists).

AMC\$EOI

File is positioned at the end-of-information.

wait: amt\$file_position

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR ost\$status

Status variable in which the completion status is returned.

Condition Identifiers

aae\$file_at_file_limit

aae\$file_is_ruined

aae\$key_not_found

aae\$major_key_too_long

aae\$nonembedded_key_not_given

aae\$not_enough_permission

Remarks

- The AMP\$START call does not specify a working storage area so the key cannot be specified in the working storage area as it can on other calls. Instead, the key_location parameter must point to the location of the key.
- If an alternate key has been selected and the key is a concatenated key, the values for the key fields are assembled at key_location. The key fields must be concatenated as defined for the key. For example, if the key is the last three bytes of the record followed by the first three bytes of the record, the value at key_location must be the last three bytes followed by the first three bytes. For more information on concatenated keys, refer to Alternate Key Definition later in this chapter.
- If no record in the file has a key that matches the specified key, a trivial error (aae\$key_not_found) occurs. The file is left positioned either at the beginning of the first record whose key is greater than the specified key or, if the specified key is greater than all keys in the file, at the end-of-information.

Reading Records

For records to be read from an indexed sequential file, the file must be open for record access with at least read access permission. However, it is recommended that the file be opened with both read and modify access permissions. Modify access permission allows access statistics to be updated without allowing any record in the file to be altered.

A read operation transfers a record from the file to the specified working storage area and positions the file at the end of the returned record. The number of bytes in the record is returned in the `record_length` parameter.

You can read records either sequentially by position or randomly by key value. A sequential read returns the next logical record in the file. A random read returns the record identified by the specified key.

Sequential Access

Records can be read sequentially from an indexed sequential file using `AMP$GET_NEXT_KEY` or `AMP$GET_NEXT` calls. Use of `AMP$GET_NEXT_KEY` calls is recommended for reading indexed sequential files. `AMP$GET_NEXT` should be used only if a common interface for writing records, regardless of file organization, is required.

`AMP$GET_NEXT_KEY` returns the key of each record in the location specified by the `key_location` parameter. The task can check the `file_position` value returned to determine when to stop reading records.

Random Access

Records are read randomly by key value using the `AMP$GET_KEY` call. To retrieve a single record from the indexed sequential file, you specify a key value, and the system returns to the working storage area the record with the matching key, if it exists.

The `major_key_length` parameter allows the `AMP$GET_KEY` call to read the first record with the specified major key. The `key_relation` parameter allows `AMP$GET_KEY` to specify the relation between the specified key and the key of the record to be read. The relation could be equal to, greater than, or greater than or equal to. This allows the task to position the file without a separate `AMP$START` call.

Random and Sequential Access

You can also read a contiguous group of records residing anywhere in the file by combining random access and sequential access. This is accomplished by issuing an `AMP$GET_KEY` to read the first record in the contiguous group, and, then, issuing `AMP$GET_NEXT_KEY` calls (or `AMP$GET_NEXT`) to sequentially read the following records.

AMP\$GET_KEY

Purpose Reads a record from an indexed sequential file by the value of the specified key.

NOTE

To allow for updating of file statistics, the file should be opened with both PFC\$READ and PFC\$MODIFY access permissions.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$GET_KEY (file_identifier, working_storage_area, working_storage_length, key_location, major_key_length, key_relation, record_length, file_position, wait, status)

Parameters

file_identifier amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

working_storage_area ^cell
Pointer to the space to which the record is copied.

working_storage_length amt\$working_storage_length
Length, in bytes, of the working storage area.

key_location ^cell
Pointer to the key of the record to be read. Set to NIL if the key is an alternate key specified in the working storage area.

major_key_length amt\$major_key_length
Length of the major key in bytes. The major key is the leftmost bytes of the key at key_location. The major key is compared to the leftmost bytes of a key.
If the value is zero, the full key length is used. Otherwise, the number of bytes specified for the major_key_length parameter must be less than or equal to the value of the key_length parameter.

key_relation: amt\$key_relation

Relationship between the key of the record and the key at key_location. The possible values are as follows:

AMC\$EQUAL_KEY

The key of the record must be equal to the key at key_location.

AMC\$GREATER_OR_EQUAL_KEY

The key of the record must be equal to the key at key_location or, if an equal key does not exist, must be the next greater key value.

AMC\$GREATER_KEY

The key of the record must be the first key value greater than the key at key_location.

record_length: VAR of amt\$max_record_length

Variable in which the number of bytes read is returned.

file_position: VAR of amt\$file_position

Variable in which the file position at completion of the read operation is returned.

AMC\$END_OF_KEY_LIST

File is positioned at the end of the key list for the alternate_key value specified on the call.

AMC\$EOR

File is positioned at end-of-record.

AMC\$EOI

File is positioned at end-of-information.

wait: ost\$wait

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR of ost\$status

Status variable in which the completion status is returned.

Condition Identifiers

aae\$file_at_file_limit
 aae\$file_is_ruined
 aae\$key_not_found
 aae\$major_key_too_long
 aae\$nonembedded_key_not_given
 aae\$not_enough_permission
 aae\$record_longer_than_wsa

Remarks

- If an alternate key is selected, the key_location parameter can point to the location of the key or it can be set to NIL. If key_location is set to NIL, AMP\$GET_KEY expects the key to be in the working storage area. The location of the key in the working storage area must match the location of the key in the record. If the alternate key is a concatenated key, each field in the concatenated key must be stored in its appropriate location in the working storage area.
- If the value of the key_relation parameter is AMC\$EQUAL_KEY and a record with the specified key value does not exist, a trivial error occurs. The file is positioned at the point where the record would be located if it existed.
- AMP\$GET_KEY returns the actual length of the record in the variable specified by the record_length parameter. If the length of the record is greater than the length of the working storage area, working_storage_length characters are returned to the working storage area and a trivial error occurs.
- Execution of the AMP\$GET_KEY call leaves the file positioned at the end of the record that was read. (AMC\$EOR or AMC\$END_OF_KEY_LIST is returned in the file_position parameter.)

AMP\$GET_NEXT_KEY

Purpose Reads the next logical record in the indexed sequential file.

NOTE

To allow for updating of file statistics, the file should be opened with both PFC\$READ and PFC\$MODIFY access permissions.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$GET_NEXT_KEY (file_identifier, working_storage_area, working_storage_length, key_location, record_length, file_position, wait, status);

Parameters **file_identifier:** amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

working_storage_area: ^cell
Pointer to the space to which the record is copied.

working_storage_length: amt\$working_storage_length
Length, in bytes, of the working storage area.

key_location: ^cell
Pointer to the space in which the record key is returned.

record_length: VAR of amt\$max_record_length
Variable in which the number of bytes read is returned.

file_position: VAR of amt\$file_position
Variable in which the position of the file at completion of the read operation is returned.

AMC\$END_OF_KEY_LIST
File is positioned at the end of a key list (can be returned only if an alternate key was selected).

AMC\$EOR
File is positioned at the end of a record. (When an alternate key is selected, indicates that the file is not at the end of a key list.)

AMC\$EOI
File is positioned at the end of the index.

wait ost\$wait

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status VAR of ost\$status

Status variable in which the completion status is returned.

Condition Identifiers

aae\$cant_position_beyond_bound
 aae\$file_at_file_limit
 aae\$file_boundary_encountered
 aae\$file_is_ruined
 aae\$nonembedded_key_not_given
 aae\$not_enough_permission
 aae\$record_longer_than_wsa
 aae\$wsa_not_given

Remarks

- AMP\$GET_NEXT_KEY returns the next record in the file as listed in the index.
- AMP\$GET_NEXT_KEY returns the file_position AMC\$EOR (or AMC\$END_OF_KEY_LIST for an alternate key) when it returns a record to the working storage area.

When AMP\$GET_NEXT_KEY reads the last record in the file, it returns AMC\$EOR (or AMC\$END_OF_KEY_LIST for an alternate key) as the file position. The next AMP\$GET_NEXT_KEY call returns AMC\$EOI as the file position; it returns no data. If the task calls AMP\$GET_NEXT_KEY again after AMC\$EOI has been returned, the condition AAESCANT_POSITION_BEYOND_BOUND is returned.

For more information on the use of this call with alternate keys, refer to Using Alternate Keys later in this chapter.

- The key will be returned as a separate character string to key_location unless the key_location parameter is set to NIL.
- At the completion of the read request, the record_length parameter is set to the length of the record that was read. If the sequential read operation was unsuccessful, the record_length parameter is not defined.
- If the length of the record that is read is greater than the length of the working storage area as specified by the working_storage_length parameter, working_storage_length characters are returned and a trivial error occurs.
- When a file is being read but not updated, the file should be opened with both PFC\$READ and PFC\$MODIFY permission. The PFC\$MODIFY permission allows access statistics to be updated without allowing any record in the file to be altered.

Replacing and Deleting Records

A call to replace or delete a record specifies the primary key of the record. Alternate keys are not used to replace or delete records; selection of an alternate key does not affect subsequent replace or delete calls.

You can replace any record in an existing file with a new record that has the same primary key value. The new record can be shorter or longer than the existing record as long as the length of the record is within the limits established by the min_record_length and max_record_length attributes. The record in the working storage area replaces the record in the file with the specified primary key value.

The AMP\$REPLACE_KEY and AMP\$PUTREP calls both replace the record having the primary key specified on the call. However, if no record in the file has a matching primary key, an AMP\$REPLACE_KEY call returns a trivial error, whereas an AMP\$PUTREP call acts as an AMP\$PUT_KEY call and writes the record to the file.

The AMP\$DELETE_KEY procedure deletes a record from an indexed sequential file. Deleting records does not release file space; the space is reused when new records are added.

AMP\$PUTREP

Purpose Replaces a record if the record exists in the indexed sequential file or adds a new record if the record does not exist.

NOTE

The file must be opened with at least PFC\$APPEND and PFC\$SHORTEN access permissions.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$PUTREP (**file_identifier**, **working_storage_area**, **working_storage_length**, **key_location**, **wait**, **status**)

Parameters **file_identifier**: amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

working_storage_area: ^cell
Pointer to the new record.

working_storage_length: amt\$working_storage_length
Length, in bytes, of the record to be written.

key_location: ^cell
Pointer to the primary key of the new record; specify NIL if primary keys are embedded.

wait: ost\$wait
Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR of ost\$status
Status variable in which the completion status is returned.

Condition Identifiers aae\$file_at_file_limit
aae\$file_at_user_record_limit
aae\$file_full_no_puts_or_reps
aae\$file_is_ruined
aae\$key_required
aae\$nonembedded_key_not_given

AMP\$REPLACE_KEY

Condition	aae\$duplicate_alternate_key
Identifiers	aae\$file_at_file_limit aae\$file_full_no_puts_or_reps aae\$file_is_ruined aae\$key_not_found aae\$key_required aae\$nonembedded_key_not_given aae\$not_enough_permission aae\$sparse_key_beyond_eor
Remarks	<ul style="list-style-type: none">• The replace request fails if the file does not contain a record whose primary key matches the primary key of the replacement record. This error is a trivial error; file processing can continue.• For AMC\$VARIABLE and AMC\$UNDEFINED type records, the new record can be smaller or larger than the existing record; however, the length of the new record must be within the minimum and maximum record length values defined for the file.• For AMC\$ANSI_FIXED type records, the value of working_storage_length is ignored and the value of the fixed record length (max_record_length) is used.• Execution of an AMP\$REPLACE_KEY call does not change the position of the file. <p>An AMP\$REPLACE_KEY call updates the alternate indexes for the new record if alternate keys are defined for the file. Calls to put or replace records are effective even if an alternate key is currently selected for reading and positioning the file.</p>

AMP\$DELETE_KEY

Purpose Removes a record from an indexed sequential file.

NOTE

The file must be open with at least PFC\$SHORTEN permission.

The procedure declaration for this procedure is stored as a deck in file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE. Therefore, you must specify this file as an alternate base library when expanding your source program decks.

Format AMP\$DELETE_KEY (**file_identifier**, **key_location**, **wait**, **status**)

Parameters **file_identifier**: amt\$file_identifier

File identifier returned by the AMP\$OPEN call for the file.

key_location: ^cell

Pointer to the primary key of the record to be deleted.

wait: ost\$wait

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR of ost\$status

Status variable in which the completion status is returned.

Condition Identifiers aae\$file_at_file_limit
 aae\$file_is_ruined
 aae\$key_required
 aae\$nonembedded_key_not_given
 aae\$not_enough_permission

- Remarks**
- When the delete request is executed, the specified record is either flagged as deleted or physically deleted from the data block. When the first record in a data block is deleted, index blocks are updated as applicable.
 - If execution of a delete request empties a data or index block, the block is linked into a chain of empty blocks. These blocks are reused when new blocks are required for file expansion.
 - If the file does not contain a record whose primary key matches the specified primary key value, a trivial error (aae\$key_not_found) occurs.
 - Execution of an AMP\$DELETE_KEY call does not change the position of the file.

An AMP\$DELETE_KEY call updates the alternate indexes if alternate keys are defined for the file. Calls to delete records are effective even if an alternate key is currently selected for reading and positioning the file.

- When deleting a series of contiguous fixed-length records, you can save execution time by beginning with the record having the highest primary key value. Deletion of the last record in a data block is performed quickly because the system just needs to reduce the record count by one. Deletion of the first record in a data block, however, can move all remaining records in the data block. By deleting records in order from the highest to the lowest primary key value, you can avoid relocation of records to be subsequently deleted.

Monitoring the Index Levels in an Indexed Sequential File

As described at the beginning of this chapter, record access in an indexed sequential file is through a hierarchy of index blocks. Each additional index level in the hierarchy requires an additional index block search for each data record access. Performance is usually best when no more than two index levels exist.

The efficiency of an indexed sequential file also depends on the block size chosen. A larger block size requires fewer index records, but it increases the time required for each block search.

An efficient indexed sequential file economizes space and the time needed to access a record. The efficiency of the file depends on the attribute values specified when the file was created. However, selecting suitable attribute values requires that you know how large the file will grow.

You should monitor the growth of an indexed sequential file by periodically checking the value of the `levels_of_indexing` file attribute. If the number of index levels remains constant, the time to retrieve a record randomly remains the same.

The `AMP$FETCH_ACCESS_INFORMATION` call can fetch the current value of the `levels_of_indexing` attribute. (`AMP$FETCH_ACCESS_INFORMATION` is described in chapter 7, *Opening and Closing Files*.)

Recreating an Indexed Sequential File

If the indexed sequential file becomes inefficient or if the current structure of the file does not match your needs, you should consider recreating the file. You can recreate the file as follows:

- Calling `AMP$COPY_FILE` as described in chapter 11, *File Copying*.
- Using the File Management Utility (FMU).

The `AMP$COPY_FILE` call recreates an indexed sequential file by copying the records in the file to another file. The existing file is called the input file; the new file is the output file. You should set the appropriate attribute values for the output file as described under *Setting File Attributes* in this chapter.

Recreating a file using the File Management Utility is described in the SCL Advanced File Management manual.

If the file has one or more alternate keys defined, you must redefine the alternate keys after the file is recreated. AMP\$COPY_FILE and FMU do not recreate the alternate key definitions. You should save the alternate key definitions from the old indexed sequential file on a text file so that you can use the definitions to redefine the alternate keys on the new file. To get the alternate key definitions used by the file, call AMP\$GET_KEY_DEFINITIONS. To redefine the alternate keys, open the new file, specify each alternate key definition on an AMP\$CREATE_KEY_DEFINITION call, and apply the definitions with an AMP\$APPLY_KEY_DEFINITIONS call.

Indexed Sequential File Example

The following example consists of two programs. The first program creates an indexed sequential file and the second program updates the file.

The first program (module CREATE) creates an indexed sequential file named INDEXED by copying records from a sequential file with local file name ORIGINAL_DATA. The first 15 characters of the record are used as the embedded primary key. The records are fixed-length records, each 55 characters long.

The following is a listing of the data in file ORIGINAL_DATA. The first column is a country name, the second is the population of the country, the third is the size of the country (in square miles), and the fourth is the capital of the country. There are several errors in the data; these will be fixed with the update program.

Algeria	19709000	919591	Algiers
Australia	14796000	2967895	Melbourne
Austria	7476000	32374	Vienna
Belgium	9875000	11781	Brussels
Canada	20050000	3851791	Montreal
Denmark	5157000	16629	Copenhagen
France	53844000	211207	Paris
Great Britain	55717000	94226	London
India	700734000	1269340	Delhi
Ireland	3349000	27136	Dublin
Ivory Coast	8513000	124503	Abidjan
Japan	118783000	143750	Yokohama
Mexico	70143000	761601	Mexico
Sweden	8335000	173731	Stockholm
Switzerland	6300000	15941	Bern
Tanzania	18744000	364898	Zanzibar
Turkey	47284000	301381	Ankara
United Kingdom	55717000	94226	London
United States	225195000	3615105	Washington
USSR	269302000	8649498	Moscow
Venezuela	15771000	352143	Caracas
West Germany	60948000	95976	Bonn

The second program (module UPDATE) adds, deletes, and replaces records in the file INDEXED created by the first program. The second program reads its input from a file named UPDATE_DATA.

The directives on file UPDATE_DATA are listed in the program output. In the program only the first letter of the words Delete, Replace, and Put are used. The full word is included in the file to make the example clearer. Only the primary key is required to delete a record.

This is a source listing of the program that creates the indexed sequential file. The program uses the common procedures listed in appendix F to inspect the status variable after each call and to produce a report on file \$OUTPUT.

```

MODULE create ;
?? left := 1, right := 110 {source line margin control} ??
?? PUSH (LIST := OFF) ??
*copyc amp$close
*copyc amp$file
*copyc amp$get_next
*copyc amp$open
*copyc amp$put_key
{ This deck contains the common procedures listed in appendix F. }
*copyc comproc
?? POP ??

{ This program creates an indexed sequential file (ISFILE) from }
{ a sequential file (DATAIN). The primary key for ISFILE is }
{ the name of the country. }

CONST
    key_length = 15,
    max_record_length = 55,
    record_count = 30,
    key_position = 0,
    data_padding = 15,
    index_padding = 10,
    index_levels = 2;

VAR
{ Declare variables for ISFILE.}
    isfile: amt$local_file_name,
    isfile_id: amt$file_identifier,
    isfile_fpos: amt$file_position,
{ Declare variables for DATAIN.}
    datain: amt$local_file_name,
    sqfile_id: amt$file_identifier,
    sqfile_fpos: amt$file_position,
    sqfile_transfer_count: amt$transfer_count,
    sqfile_byte_address: amt$file_byte_address,
{ Wsa is used by both ISFILE and DATAIN.}
    wsa: string (max_record_length);

```

```
{ Establish for file_description an array of file attribute }
{ values.}
```

```
VAR file_description: [STATIC] array [1 .. 13] of
  amt$file_item :=
    [[amc$file_organization, amc$indexed_sequential],
     [amc$max_record_length, max_record_length],
     [amc$record_type, amc$ansi_fixed],
     [amc$average_record_length, max_record_length],
     [amc$embedded_key, TRUE],
     [amc$key_length, key_length],
     [amc$key_position, key_position],
     [amc$key_type, amc$uncollated_key],
     [amc$data_padding, data_padding],
     [amc$index_padding, index_padding],
     [amc$index_levels, index_levels],
     [amc$estimated_record_count, record_count],
     [amc$message_control, $amt$message_control]
     [amc$trivial_errors, amc$messages, amc$statistics]]];

PROGRAM creation_phase (VAR program_status : ost$status) ;

  p#start_report_generation('Begin indexed sequential file
    creation.');
```

```
  isfile := 'indexed';
  datain := 'original_data';
  amp$file (isfile, file_description, status);
  p#inspect_status_variable ;

  amp$open (isfile, amc$record, NIL, isfile_id, status);
  p#inspect_status_variable ;
  amp$open (datain, amc$record, NIL, sqfile_id, status);
  p#inspect_status_variable ;
```

```
{ The next part of the program reads records from DATAIN and }
{ writes the records to ISFILE. A WHILE loop is used to read }
{ and write the records until the file position of DATAIN is }
{ end-of-information.}
```

```
wsa := ' ';
amp$get_next (sqfile_id, ^wsa, max_record_length,
             sqfile_transfer_count, sqfile_byte_address,
             sqfile_fpos, status);
p#inspect_status_variable;

WHILE (sqfile_fpos <> amc$eoi) DO
  {The working storage length (the third parameter) is
  {ignored because the record type is amc$ansi_fixed.
  amp$put_key (isfile_id, ^wsa, 0, NIL, osc$wait, status);
  p#inspect_status_variable;
  wsa := ' ';
  amp$get_next (sqfile_id, ^wsa, max_record_length,
               sqfile_transfer_count, sqfile_byte_address,
               sqfile_fpos, status);
  p#inspect_status_variable;
WHILEND;

amp$close (isfile_id, status);
p#inspect_status_variable;
amp$close (sqfile_id, status);
p#inspect_status_variable;

p#stop_report_generation('Indexed sequential file
  creation complete. ');
program_status.normal := TRUE ;
{ Exit with normal status. }

PROCEND creation_phase ;

MODEND create;
```

This is a source listing of the program that updates the indexed sequential file. The program uses the common procedures listed in appendix G to inspect the status variable after each call and to produce a report on file \$OUTPUT.

```

MODULE update;
?? left := 1, right := 110 {source line margin control} ??
?? PUSH (LIST:=OFF) ??
*copyc amp$close
*copyc amp$delete_key
*copyc amp$get_next
*copyc amp$open
*copyc amp$putrep
{ The COMPROC deck contains the common procedures listed in }
{ appendix G. }
*copyc comproc
?? POP ??

{ This program updates indexed sequential file (INDEXED) }
{ information contained in an update file (UPDATE_DATA). }

CONST
    record_length = 55;

VAR
{ Declare variables for ISFILE.}
    isfile: amt$local_file_name := 'indexed',
    isfile_id: amt$file_identifier,
    isfile_fpos: amt$file_position,
    key: string (15),
    isfile_wsa: string (record_length),

{ Declare variables for UPDATE.}
    update: amt$local_file_name := 'update_data',
    update_id: amt$file_identifier,
    update_fpos: amt$file_position,
    update_transfer_count: amt$transfer_count,
    update_byte_address: amt$file_byte_address,
    update_wsa: string (record_length + 7),

{ Declare access_selections array for amp$open.}
    access_selections: [STATIC] array [1 .. 1] of amt$file_item
        := [[amc$message_control, $amt$message_control
            [amc$trivial_errors, amc$messages, amc$statistics]]];

```

INDEXED SEQUENTIAL FILE EXAMPLE

```

PROGRAM updating_phase (VAR program_status : ost$status) ;

    p#start_report_generation('Begin file update.');
```

amp\$open (isfile, amc\$record, ^access_selections,
isfile_id, status);
p#inspect_status_variable;
amp\$open (update, amc\$record, NIL, update_id, status);
p#inspect_status_variable;

```

{ The WHILE loop that follows reads an update record from UPDATE }
{ and edits ISFILE accordingly. The update information is         }
{ contained in the first 7 characters of the records in UPDATE;   }
{ however, only the first character is used to determine          }
{ whether a delete, put, or replace operation is to be           }
{ performed. If the operation requested is not a delete, put, or }
{ replace, a message and the update record are printed on the    }
{ output listing. If the status parameter check shows that an   }
{ error occurred, then control is returned to the system.}

    update_wsa := ' ';
    amp$get_next (update_id, ^update_wsa, STRLENGTH(update_wsa),
        update_transfer_count, update_byte_address, update_fpos,
        status);
    p#inspect_status_variable;
    WHILE (update_fpos <> amc$eoi) DO
        p#put_m (TRUE, update_wsa(1, update_transfer_count));
        isfile_wsa := update_wsa (8, *);
        key := isfile_wsa (1, 15);
        CASE update_wsa (1) OF
            = 'D' =
                amp$delete_key (isfile_id, ^key, osc$wait, status);
                p#inspect_status_variable ;
            = 'P', 'R' =
                amp$putrep (isfile_id, ^isfile_wsa, 0, NIL, osc$wait, status);
                p#inspect_status_variable;
            ELSE
                p#put_m (FALSE, 'Invalid code given as first character. -->');
                p#put_m (TRUE , update_wsa(1, update_transfer_count));
        CASEEND;
        update_wsa (1, * ) := ' ';
        amp$get_next (update_id, ^update_wsa, STRLENGTH(update_wsa),
            update_transfer_count, update_byte_address,
            update_fpos, status);
        p#inspect_status_variable;
    WHILEEND;
```

```
amp$close (isfile_id, status);  
  p#inspect_status_variable;  
amp$close (update_id, status);  
  p#inspect_status_variable;
```

```
p#stop_report_generation('File update complete.');
```

```
program_status.normal := TRUE ;  
{ Exit with normal status. }
```

```
PROCEND updating_phase ;
```

```
MODEND update;
```

INDEXED SEQUENTIAL FILE EXAMPLE

Assuming the program source text is stored as decks CREATE and UPDATE on the source library \$USER.MY_LIBRARY and the data files are stored in the \$USER catalog, the following are the SCL commands required to expand, compile, attach the data files, and execute the programs. After the commands is a listing of the update statistic messages from the programs.

```
/scu_expand_deck base=$user.my_library deck=(create,update) ..  
../alternate_base=($system.cybil.osf$program_interface, ..  
../$system.common.psf$external_interface_source)  
/cybil input=compile  
/attach_file $user.original_data  
/attach_file $user.update_data  
/lgo
```

Begin indexed sequential file creation.

```
-- File INDEXED : 0 DELETE_KEYS done since last open.  
-- File INDEXED : 0 GET_KEYS done since last open.  
-- File INDEXED : 0 GET_NEXT_KEYS done since last open.  
-- File INDEXED : 22 PUT_KEYS (and PUTREPs->put) since last open.  
-- File INDEXED : 0 PUTREPs done since last open.  
-- File INDEXED : 0 REPLACE_KEYS (and PUTREPs->replace) since last open.  
No error has been found by the program.  
Indexed sequential file creation complete.
```

Begin file update.

ReplaceCanada	24336000	3851791	Ottawa
Put China	1053788000	3705390	Beijing
Delete Great Britain			
Put Spain	38686000	194897	Madrid
Put Italy	57513000	116303	Rome
ReplaceJapan	11878300	143750	Tokyo

```
-- File INDEXED : 1 DELETE_KEYS done since last open.  
-- File INDEXED : 0 GET_KEYS done since last open.  
-- File INDEXED : 0 GET_NEXT_KEYS done since last open.  
-- File INDEXED : 3 PUT_KEYS (and PUTREPs->put) since last open.  
-- File INDEXED : 5 PUTREPs done since last open.  
-- File INDEXED : 2 REPLACE_KEYS (and PUTREPs->replace) since last open.  
No error has been found by the program.  
File update complete.
```


Alternate Keys

Records within an indexed sequential file can always be accessed by their primary key values. An alternate key provides an additional way to access records.

An alternate key is a field or group of fields in a data record. Alternate key fields can overlap each other and the primary key.

Unlike a primary key value, one alternate key value be associated with several records in a file. This is because an alternate key value need not be unique. The same alternate key value can occur in several records (such as the same job title associated with many names).

A record can contain more than one alternate key value if the alternate key is defined as a field that repeats in the record; thus, a single record could contain several alternate key values (such as the license numbers of several cars owned by one person).

Although alternate keys can be used to read records or to position a file, they cannot be used to write, replace, or delete records. Primary keys must be used to write, replace, or delete records. (You can use alternate keys to locate the primary key of a record prior to writing, replacing, or deleting the record.)

Alternate key definitions can be created or deleted by any user of the file who has read, append, modify, and shorten access permissions. Only read permission is needed to retrieve data records or index entries.

The Alternate Index

The index blocks for the primary key were described earlier in this chapter. Each alternate key defined for a file has its own index.

An alternate index contains records that associate each alternate key value with the primary key values of the records containing that alternate key value. The list of primary key values associated with an alternate key value is its key list.

When you select an alternate key and then specify an alternate key value, the system searches for the value in the alternate index. If it finds the alternate key value, it uses the primary key values in the key list to access the records.

When one or more alternate keys are defined for a file, file updates require more time because the alternate indexes must also be updated. Alternate keys should be used only when the additional record access capability offsets the cost of increased time spent for file updates.

Alternate Key Definition

An alternate key is specified by an alternate key definition. The alternate key definition specifies the attributes of the alternate key.

Alternate keys have both required and optional attributes. The required attributes are `key_name`, `key_position`, and `key_length`. An alternate key has a name so that it can be selected for use. The alternate key position and length define the alternate key field within the record.

The key type of an alternate key specifies the order of the index records in the alternate key index. (The index records are ordered by the alternate key value in the record.) The valid key type values for an alternate key are the same as the key type values for the primary key as described earlier in this chapter. If the key type is `AMC$COLLATED_KEY`, you can explicitly specify a collation table for the alternate key or use, as the default, the collation table specified for the primary key. The default key type for an alternate key is the same as the default key type for the primary key.

Duplicate Key Values

The `duplicate_key_control` attribute controls the handling of duplicate alternate key values in an alternate index. The attribute values are:

`AMC$NO_DUPLICATES_ALLOWED`

An attempt to add a duplicate key value to the alternate index results in a trivial error.

If the duplicate is found during creation of the alternate index, creation of the alternate index is restarted using the `AMC$ORDERED_BY_PRIMARY_KEY` attribute value. (If this is not desired, set the `error_limit` attribute to 1. The occurrence of a trivial error [such as a duplicate key value] causes the trivial error limit to be reached and issuance of a fatal error. The fatal error terminates alternate index creation. No alternate indexes are created by the terminated `AMP$APPLY_KEY_DEFINITIONS` procedure; however, it does perform all pending alternate key deletions.)

If the duplicate is found during an attempt to write a record to the file, the record is not written to the file and a trivial error (`AAE$DUPLICATE_ALTERNATE_KEY`) is returned.

`AMC$ORDERED_BY_PRIMARY_KEY`

Duplicate key values are allowed in the alternate index. When a duplicate alternate key value is found, the associated primary key value is stored in the key list for the alternate key value. The primary key values are stored in sorted order according to the primary key type.

AMC\$FIRST_IN_FIRST_OUT

Duplicate key values are allowed in the alternate index. When a duplicate alternate key value is found, the associated primary key value is stored in the key list for the alternate key value. The primary keys are stored in the key list in the order the values are added to the key list, instead of in sorted order; new values are always added to the end of the key list.

AMC\$FIRST_IN_FIRST_OUT cannot be used for keys having repeating groups.

A key list ordered by AMC\$FIRST_IN_FIRST_OUT is not reordered when a data record is replaced unless the replaced record has a different alternate key value (before collation for collated keys). For example, if the first data record referenced by a key list is replaced and its alternate key value remains the same, its position in the key list remains the same; it is still the first record in the key list. However, if the alternate key value in the record has changed, the record is removed from its former key list and placed at the end of the key list for its new alternate key value; it would then be the last record in that key list.

If you do not set a value for `duplicate_key_control`, the value AMC\$NO_DUPLICATES_ALLOWED is used.

If you choose to allow duplicate alternate key values, AMC\$ORDERED_BY_PRIMARY_KEY is more efficient than AMC\$FIRST_IN_FIRST_OUT because the primary key order allows more direct index searches.

To illustrate, figure 10-6 shows a list of data records and three alternate indexes having each of the three `duplicate_key_control` attribute values. The primary key is defined as the employee number; the last name, full name, and job title are defined as alternate keys. The alternate index for the last name uses AMC\$ORDERED_BY_PRIMARY_KEY; the alternate index for the full name uses AMC\$NO_DUPLICATES_ALLOWED; and the alternate index for job title uses AMC\$FIRST_IN_FIRST_OUT.

The key list for Computer Operator in the AMC\$FIRST_IN_FIRST_OUT index shows that the record with primary key value 38602 was written to the file after the record for primary key value 75090.

ALTERNATE KEYS

Data Records:				
<u>Emp. #</u>	<u>Last Name</u>	<u>First Name</u>	<u>Job Title</u>	
13905	McGuire	Stewart	Clerk	
38602	Johnstone	Mark	Computer Operator	
39048	Jetson	Harry	Asst Manager	
39248	Miller	Robert	Driver	
42976	Stevens	Carol	Manager	
49257	Lane	Gladys	Accountant	
51234	Miller	Catherine	Secretary	
75090	Arnold	Terry	Computer Operator	
82176	Beirmeyer	William	Driver	

Alternate Indexes:					
<u>Last Name</u>		<u>Full Name</u>			
(AMC\$ORDERED_BY_PRIMARY KEY)		(AMC\$NO_DUPLICATES_ALLOWED)			
Arnold	75090	Arnold	Terry	75090	
Beirmeyer	82176	Beirmeyer	William	82176	
Jetson	39048	Jetson	Harry	39048	
Johnstone	38602	Johnstone	Mark	38602	
Lane	49257	Lane	Gladys	49257	
McGuire	13905	McGuire	Stewart	13905	
Miller	39248	51234	Miller	Catherine	51234
Stevens	42976	Miller	Robert	39248	
		Stevens	Carol	42976	

<u>Job Title</u>	
(AMC\$FIRST_IN_FIRST_OUT)	
Accountant	49257
Asst Manager	39048
Clerk	13905
Computer Operator	75090 38602
Driver	39248 82176
Manager	42976
Secretary	51234

Figure 10-6. Example of Duplicate Key Control

Null Suppression

The `null_suppression` attribute allows you to exclude from an alternate index all records that have a null value for the specified alternate key. Null suppression can save space, access time, and update time because the index is smaller when the null alternate key values are excluded.

If the key type is `AMC$INTEGER_KEY`, the null value is 0; if the key type is `AMC$UNCOLLATED_KEY`, the null value is all spaces; if the key type is `AMC$COLLATED_KEY`, the null value is all spaces before collation.

If null suppression is not specified, records containing a null value in the alternate key field are indexed by the null value. The records can later be accessed by specifying the null value as the alternate key value.

For example, consider the records shown in figure 10-7. Each record consists of an employee number, name, job title, and car license number. Assume the primary key is the employee number and the license number is the alternate key for which `AMC$ORDERED_BY_PRIMARY_KEY`. The first alternate index is the index created if null suppression is not used. The second alternate index is the index created if null suppression is used.

Data Records:				
<u>Emp. #</u>	<u>Last Name</u>	<u>First Name</u>	<u>Job Title</u>	<u>License #</u>
39248	Miller	Robert	Driver	3BMW862
42976	Stevens	Carol	Manager	1BOS003
39048	Jetson	Harry	Asst Manager	
51234	Miller	Catherine	Secretary	
82176	Beirmeyer	William	Driver	3CAR395
75090	Arnold	Terry	Computer Operator	4CI0999
49257	Lane	Gladys	Accountant	1CPA120
38602	Johnstone	Mark	Computer Operator	
13905	McGuire	Stewart	Clerk	5PEN485

Alternate Index Without Null Suppression:		Alternate Index With Null Suppression:	
<u>License #</u>	<u>Emp. #</u>	<u>License #</u>	<u>Emp. #</u>
	39048	1BOS003	42976
1BOS003	42976	1CPA120	49257
1CPA120	49257	3BMW862	39248
3BMW862	39248	3CAR395	82176
3CAR395	82176	4CI0999	75090
4CI0999	75090	5PEN485	13905
5PEN485	13905		

Figure 10-7. Example of Null Suppression

Sparse Keys

Sparse key control is used to determine whether a record is included or excluded in an alternate index. To use sparse key control, you specify three values:

A one-character field within the minimum record length (`sparse_key_control_position`).

One or more possible sparse control values (`sparse_key_control_characters`).

Whether the alternate key value should be included or excluded if the character in the sparse key field matches one of the specified values (`sparse_key_control_effect`).

If sparse key control is specified for an alternate key, the alternate key field or fields need not be within the minimum record length. If the character at the `sparse_key_control_position` indicates that the record should be included in the alternate index, but the record has no alternate key value because the record ends before the alternate key field, the record is not included in the alternate index. Although the record is not included in the alternate index, it is written to the file and a trivial error (`AAE$SPARSE_KEY_BEYOND_EOR`) is returned.

To illustrate sparse key control use, figure 10-8 contains a list of data records and two alternate indexes. The data records contain an employee number, name, job title, pay rate, and pay period. Assume that the employee number is the primary key and that you want to define pay rate as an alternate key (using `AMC$ORDERED_BY_PRIMARY_KEY`). However, hourly and monthly pay rates are not directly comparable so two alternate keys are defined, one to access records with an hourly pay rate and the other to access records with a monthly pay rate. Both alternate keys specify the pay rate field as the alternate key field and the pay period field as the `sparse_key_control_position`. The hourly pay rate alternate key specifies H for the `sparse_key_control_characters` attribute; the monthly pay rate alternate key specifies M. Both alternate keys specify that the record is to be included if the sparse key matches.

Data Records:

<u>Emp. #</u>	<u>Last Name</u>	<u>First Name</u>	<u>Job Title</u>	<u>Pay Rate</u>	<u>Pay Period</u>
39248	Miller	Robert	Driver	10.00	H
42976	Stevens	Carol	Manager	10000.00	M
39048	Jetson	Harry	Asst Manager	9000.00	M
51234	Miller	Catherine	Secretary	9.00	H
82176	Beirmeyer	William	Driver	10.00	H
75090	Arnold	Terry	Computer Operator	7.00	H
49257	Lane	Gladys	Accountant	8000.00	M
38602	Johnstone	Mark	Computer Operator	6.00	H
13905	McGuire	Stewart	Clerk	6.00	H

Alternate Indexes:

<u>Monthly Pay</u>	<u>Emp. #</u>	<u>Hourly Pay</u>	<u>Emp. #</u>
8000.00	49257	6.00	13905 38602
9000.00	39048	7.00	75090
10000.00	42976	9.00	51234
		10.00	39248 82176

Figure 10-8. Sparse Key Example

Concatenated Keys

A concatenated key is an alternate key formed from several fields in the record. The fields can be noncontiguous. This means that you can set up the fields in the data records in any order and still use any combination of fields as an alternate key. The order in which the fields are concatenated to form the key is specified when the key is defined.

A concatenated key can comprise up to 64 fields. Each field can be a different key type. All `AMC$COLLATED_KEY` fields use the same collation table.

The leftmost field of the key (the most significant field) is specified by the `key_position`, `key_length`, and `key_type` attributes. Each concatenated field is specified by a set of three attribute values: `concatenated_key_position`, `concatenated_key_length`, and `concatenated_key_type`. The order of the concatenated fields within the key is specified by the order you specify the fields in the alternate key definiton.

A concatenated key can use sparse key control and/or null suppression. A concatenated key is considered to have a null value if the values in all fields of the key are null (before collation for collated keys).

Figure 10-9 shows a concatenated key example. Each data record contains an employee number, name, and job title. Assume that the employee number is the primary key, and that the employee first name and last name form a concatenated alternate key. The resulting alternate index is shown.

Data Records:			
<u>Emp. #</u>	<u>Last Name</u>	<u>First Name</u>	<u>Job Title</u>
39248	Miller	Robert	Driver
42976	Stevens	Carol	Manager
39048	Jetson	Harry	Asst Manager
51234	Miller	Catherine	Secretary
82176	Beirmeyer	William	Driver
75090	Arnold	Terry	Computer Operator
49257	Lane	Gladys	Accountant
38602	Johnstone	Mark	Computer Operator
13905	McGuire	Stewart	Clerk

Alternate Index:		
Carol	Stevens	42976
Catherine	Miller	51234
Gladys	Lane	49257
Harry	Jetson	39048
Mark	Johnstone	38602
Robert	Miller	39248
Stewart	McGuire	13905
Terry	Arnold	75090
William	Beirmeyer	82176

Figure 10-9. Concatenated Key Example

Repeating Groups

The repeating groups attribute allows a data record to contain more than one alternate key value. This allows the same primary key value to be associated with more than one alternate key value in an alternate index.

To specify an alternate key field within a repeating group, you specify values for the following fields:

`key_position`: the beginning of the first alternate key value in a record. (Bytes are numbered from the left beginning with 0.)

`key_length`: the length of each alternate key value.

`repeating_group_length`: the length of the repeating group, that is, the distance between the beginning of an alternate key value and the beginning of the next alternate key value.

`repetition_control.repeat_to_end_of_record`: indicates whether the repeating group repeats a fixed number of times or until the end of the record.

If `repetition_control.repeat_to_end_of_record` is true, the repeating group repeats until the end of the record. In this case, the alternate key values need not occur within the minimum record length. The system stores as many alternate key values as the record length allows; it ignores trailing information not long enough to be a repeating group.

If `repetition_control.repeat_to_end_of_record` is false, the alternate key field repeats a fixed number of times. In this case, the alternate key values must occur within the minimum record length. The number of alternate key values in each record is specified by `repetition_control.repeating_group_count`.

If an alternate key value appears more than once in a record, the primary key value is stored only once in the key list for that alternate key value. This is illustrated for the name Darryl in the figure 10-10 example. Even though employee 51234 has two dependents named Darryl, the primary key value is stored only once for alternate key value Darryl.

Repeating groups cannot be used with concatenated keys or when `duplicate_key_control` is set to `AMC$FIRST_IN_FIRST_OUT`.

As an example of repeating groups, consider the data records shown in figure 10-10. Each record contains an employee number, name, job title, and a list of employee children. Each child is identified by two fields giving his or her first name and age. Assume that the employee number is the primary key and that the children's names are to be defined as the alternate key.

ALTERNATE KEYS

To do so, the two fields identifying each child are defined as a repeating group. The key_position is specified as the beginning of the first child's name in the record. The key_length is the length of the child's name field. The repeating_group_length is the distance from the beginning of one child's name to the beginning of the next child's name in the record. The repeat_to_end_of_record field is true because the number of children is variable. The resulting index is shown.

Data Records:

<u>Emp. #</u>	<u>Last Name</u>	<u>First Name</u>	<u>Children</u>
39248	Miller	Robert	Mary 03Thomas 01
42976	Stevens	Carol	Mary 18George 13Richard 07
39048	Jetson	Harry	Patricia 10
51234	Miller	Catherine	Larry 06Darryl 05Darryl 05
82176	Beirmeyer	William	Linda 14
75090	Arnold	Terry	Anne 06Tara 04
49257	Lane	Gladys	Darryl 19
38602	Johnstone	Mark	Larry 08Linda 06Lily 05
13905	McGuire	Stewart	Mary 02

Alternate Index:

<u>Dependent</u>	<u>Emp. #</u>
Anne	75090
Darryl	49257 51234
George	42976
Larry	38602 51234
Lily	38602
Linda	38602 82176
Mary	13905 39248 42976
Patricia	39048
Richard	42976
Tara	75090
Thomas	39248

Figure 10-10. Repeating Groups Example

Creating and Deleting Alternate Keys

You can create and delete alternate keys in a new indexed sequential file or an existing indexed sequential file. To do so, perform these steps:

1. Open the indexed sequential file, if it is not already open.
2. Issue an AMP\$CREATE_KEY_DEFINITION call for each alternate key to be created. Issue an AMP\$DELETE_KEY_DEFINITION call for each alternate key to be deleted.
3. To implement the alternate key definitions and deletions specified in step 2, issue an AMP\$APPLY_KEY_DEFINITIONS call. Or, to discard the specified definitions and deletions, issue an AMP\$ABANDON_KEY_DEFINITIONS call.

Although you can define alternate keys before any records are written to the file, it is more efficient to define alternate keys after the initial records are written to the file. This is because the alternate index can then be written in sorted order. If the alternate index is written as each record is written, the alternate index is written in random order. This takes much longer. The efficiency difference is even greater when the file has more than one alternate index.

Applying an alternate key definition to a file can require considerable processing time if the file is large because creation of the new alternate index requires that all records in the file be read.

AMP\$CREATE_KEY_DEFINITION

Purpose	Defines an alternate key.
Format	AMP\$CREATE_KEY_DEFINITION (file_identifier , key_name , key_position , key_length , optional_attributes , status)
Parameters	<p>file_identifier: amt\$file_identifier File identifier returned by the AMP\$OPEN call for the file.</p> <p>key_name: amt\$key_name Alternate key name. The name is specified as a 31-character string; the name is left-justified with blank fill within the string and must follow the SCL naming rules.</p> <p>key_position: amt\$key_position Position of the first byte of the alternate key in the record. (The bytes in a record are numbered from the left, beginning with zero.)</p> <p>key_length: amt\$key_length Length, in bytes, of the alternate key. The maximum length is 255 bytes.</p> <p>optional_attributes: ^amt\$optional_key_attributes Pointer to an adaptable array defining optional attributes of the alternate key. Specify NIL if no optional attributes are to be specified. Each record in the array specifies an optional attribute; the attribute defined is indicated by the SELECTOR field of the record. Table 10-1 lists the SELECTOR field values and the attribute record fields generated for each SELECTOR field value.</p> <p>status: VAR of ost\$status Status variable in which the completion status is returned.</p>
Condition Identifiers	<p>aae\$alt_key_past_minrl</p> <p>aae\$bad_name</p> <p>aae\$cant_create_existing_name</p> <p>aae\$concatenated_key_too_big</p> <p>aae\$cant_get_collate_table</p> <p>aae\$collated_altkey_no_table</p> <p>aae\$no_repeating_group</p>

Remarks

- A subsequent AMP\$APPLY_KEY_DEFINITIONS call is required to implement an alternate key definition specified by an AMP\$CREATE_KEY_DEFINITION call. Before the apply operation, an alternate key definition is only pending and cannot be used to access records in the file. A call to AMP\$ABANDON_KEY_DEFINITIONS discards pending alternate key definitions.
- If the selector field in a record in the optional_attributes array has the value AMC\$NULL_ATTRIBUTE, the record is ignored.
- Sparse key control is defined by the sparse_key_control_position, sparse_key_control_characters, and sparse_key_control_effect values. If an alternate key is subject to sparse key control, the sparse key control character must be within the minimum record length, but the alternate key fields need not be. For more information, see the sparse keys description earlier in this chapter.
- A concatenated key can comprise up to 64 fields in the record. The most significant field of the key is defined by the key_position and key_length values. Each field concatenated to the first field is specified by a record in the optional_attributes array containing concatenated_key_position, concatenated_key_length, and concatenated_key_type fields. The order in which the fields are concatenated corresponds to the order of the records in the array.

The total length of a concatenated key can be a maximum of 700 bytes.

- The first alternate key value in a repeating group begins at key_position. Subsequent keys are found by adding the value of repeating_group_length to key_position until either the repeating_group_count is satisfied (repeat_to_end_of_record is FALSE) or the end of the record is reached (repeat_to_end_of_record is TRUE).
- Repeating groups cannot be used with concatenated keys. Also, repeating groups cannot be used when duplicate_key_control is set to AMC\$FIRST_IN_FIRST_OUT.

NOTE

The CYBIL declaration for AMT\$OPTIONAL_KEY_ATTRIBUTE in appendix C lists additional fields besides those listed in table 10-1. These additional fields are for features not yet implemented.

Table 10-1. Optional Attribute Record Contents (AMT\$OPTIONAL_KEY_ATTRIBUTE)

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC\$KEY_TYPE	<p>KEY_TYPE : amt\$key_type Type of the alternate key.</p> <p>AMC\$UNCOLLATED_KEY Keys (1 through 255 bytes) ordered byte-by-byte according to the ASCII character set sequence (listed in appendix B). The key can be a positive integer or a string of ASCII character codes.</p> <p>AMC\$INTEGER_KEY Integer keys (1 through 255 bytes) ordered numerically. The integer can be positive or negative.</p> <p>AMC\$COLLATED_KEY Keys (1 through 255 bytes) ordered according to a user-specified collation table (see the COLLATE_TABLE_NAME description in this table).</p> <p>If you omit the attribute, AMC\$UNCOLLATED_KEY is used.</p>
AMC\$COLLATE_TABLE_NAME	<p>COLLATE_TABLE_NAME : pmt\$program_name</p> <p>Name of the collation table to be used for collating the alternate key. (The alternate key collation table can differ from the primary key collation table. See appendix E for more information on collation tables.)</p> <p>If you omit the attribute and the key type of both the alternate key and the primary key is AMC\$COLLATED_KEY, the collation table for the primary key is used; however, if the alternate key type is AMC\$COLLATED_KEY but the primary key type is not AMC\$COLLATED_KEY, you must specify a collation table for the alternate key.</p>

(Continued)

Table 10-1. Optional Attribute Record Contents (AMT\$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC\$DUPLICATE_KEYS	<p>DUPLICATE_KEY_CONTROL : amt\$duplicate_key_control</p> <p>Indicates how duplicate alternate key values are handled in the alternate index.</p> <p>AMC\$NO_DUPLICATES_ALLOWED No duplicate alternate key values are allowed in the alternate index.</p> <p>AMC\$FIRST_IN_FIRST_OUT Duplicate alternate key values are ordered according to when the record is written to the file.</p> <p>AMC\$ORDERED_BY_PRIMARY_KEY Duplicate alternate key values are ordered according to primary key values.</p> <p>Omission causes AMC\$NO_DUPLICATES_ALLOWED to be used.</p>
AMC\$NULL_SUPPRESSION	<p>NULL_SUPPRESSION : boolean</p> <p>Indicates whether alternate keys with a null value should be included in the alternate key index. (For AMC\$INTEGER_KEY, the null value is zero; for AMC\$UNCOLLATED_KEY, the null value is all spaces; for AMC\$COLLATED_KEY, the null value is all spaces before collation.)</p> <p>FALSE All values are included in the index.</p> <p>TRUE Null values are not included in the index.</p> <p>Omission causes FALSE to be used.</p>

(Continued)

Table 10-1. Optional Attribute Record Contents (AMT\$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC\$SPARSE_KEYS	<p data-bbox="423 282 885 336">SPARSE_KEY_CONTROL_POSITION : amt\$key_position</p> <p data-bbox="423 350 953 461">Position of the sparse key control character. The position must be within the minimum record length. (Bytes in a record are numbered from the left, beginning with zero.)</p> <p data-bbox="423 488 743 542">SPARSE_KEY_CONTROL_CHARACTERS : set of char</p> <p data-bbox="423 557 894 610">Set of characters with which the sparse key character is compared.</p> <p data-bbox="423 638 860 691">SPARSE_KEY_CONTROL_EFFECT : amt\$sparse_key_control_effect</p> <p data-bbox="423 706 953 781">Indicates whether a sparse key control character match causes the alternate key to be included or excluded from the alternate index.</p> <p data-bbox="456 808 820 833">AMC\$INCLUDE_KEY_VALUE</p> <p data-bbox="456 847 860 901">Alternate key value is included in the alternate index.</p> <p data-bbox="456 928 825 953">AMC\$EXCLUDE_KEY_VALUE</p> <p data-bbox="456 967 901 1021">Alternate key value is not included in the alternate index.</p>
AMC\$REPEATING_GROUP	<p data-bbox="423 1044 816 1097">REPEATING_GROUP_LENGTH : amt\$max_record_length,</p> <p data-bbox="423 1112 953 1219">Length, in bytes, of the repeating group of fields. It is the distance from the beginning of an alternate key value to the beginning of the next alternate key value in the record.</p>

(Continued)

Table 10-1. Optional Attribute Record Contents (AMT\$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
	<p>REPETITION_CONTROL : amt\$repetition_control</p> <p>This record indicates whether the alternate key repeats until the end of the record. If no values are specified for the repetition_control record, it is assumed that the repeating group repeats until the end of the record.</p> <p>REPEAT_TO_END_OF_RECORD : boolean</p> <p>TRUE</p> <p>The alternate key repeats until the record ends. (An incomplete key at the end of the record is not used.)</p> <p>FALSE</p> <p>The alternate key repeats the number of times specified in the REPEATING_GROUP_COUNT field. If sparse key control is not used, the specified number of key values must be within the minimum record length.</p> <p>REPEATING_GROUP_COUNT: amt\$max_repeating_group_count</p> <p>Number of times the group of fields repeats in a record. This field is generated only if REPEAT_TO_END_OF_RECORD is FALSE.</p>
<p>AMC\$CONCATENATED_KEY_PORTION</p>	<p>CONCATENATED_KEY_POSITION : amt\$key_position</p> <p>Position of a field to be concatenated to the key. (Bytes are numbered from the left, beginning with zero.)</p> <p>CONCATENATED_KEY_LENGTH : amt\$key_length</p> <p>Length, in bytes, of a field to be concatenated to the key.</p>
	<p>CONCATENATED_KEY_TYPE : amt\$key_type</p> <p>Key type of a field to be concatenated to the key. The key types are the same as for the key_type parameter.</p>

AMP\$DELETE_KEY_DEFINITION

Purpose	Requests removal of an alternate key definition by the next AMP\$APPLY_KEY_DEFINITIONS call.
Format	AMP\$DELETE_KEY_DEFINITION (file_identifier, key_name,status);
Parameters	<p>file_identifier: amt\$file_identifier File identifier returned by the AMP\$OPEN call for the file.</p> <p>key_name: amt\$key_name Name of the alternate key to be deleted. The name is specified as a 31-character string; the name is left-justified with blank fill within the string.</p> <p>status: VAR of ost\$status Status variable in which the completion status is returned.</p>
Condition Identifiers	<p>aae\$bad_name</p> <p>aae\$cant_delete_missing_name</p> <p>aae\$no_delete_current_key</p> <p>aae\$not_enough_permission</p>
Remarks	<ul style="list-style-type: none"> • A subsequent AMP\$APPLY_KEY_DEFINITIONS call is required to implement an alternate key deletion specified by an AMP\$DELETE_KEY_DEFINITION call. <p>Before the apply operation, an alternate key deletion is only pending; the alternate key remains in the file, although it is not available for use. (An instance of open that has already selected the alternate key can continue to use it; however, no instance of open can select the key while its deletion is pending.)</p> <p>A call to AMP\$ABANDON_KEY_DEFINITIONS discards pending alternate key deletions.</p> <ul style="list-style-type: none"> • You cannot delete an alternate key while you have the key selected. Before calling AMP\$DELETE_KEY_DEFINITION for the current key, you must call AMP\$SELECT_KEY to select another key; otherwise AMP\$DELETE_KEY_DEFINITION returns the condition AAE\$NO_DELETE_CURRENT_KEY.

AMP\$APPLY_KEY_DEFINITIONS

Purpose Applies the pending alternate key definitions and deletions to the file.

Format AMP\$APPLY_KEY_DEFINITIONS (file_identifier, status)

Parameters **file_identifier:** amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

status: VAR of ost\$status
Status variable in which the completion status is returned.

Condition Identifiers aae\$begin_altkey_labels
aae\$begin_delete_keydefs
aae\$duplicate_alternate_key
aae\$enable_altkey_duplicates
aae\$end_altkey_labels
aae\$end_delete_keydefs
aae\$index_being_built
aae\$keydef_has_been_deleted
aae\$no_definitions
aae\$not_enough_permission
aae\$sparse_key_beyond_eor
aae\$unexpected_dup_encountered

Remarks

- An AMP\$APPLY_KEY_DEFINITIONS call first deletes each alternate index specified by a pending alternate key deletion. It then creates an alternate index for each pending alternate key definition. A pending definition or deletion is one requested by an AMP\$CREATE_KEY_DEFINITION or AMP\$DELETE_KEY_DEFINITION call that has not yet been discarded or applied to the file. (Closing the file or issuing an AMP\$ABANDON_KEY_DEFINITIONS call discards all pending definitions and deletions.)
- If AMC\$NO_DUPLICATES_ALLOWED is specified for a new key and the file contains data, AMP\$APPLY_KEY_DEFINITIONS returns a trivial error (condition AAE\$UNEXPECTED_DUP_ENCOUNTERED) if it finds a duplicate alternate key value. It then changes the duplicate control for the index from AMC\$NO_DUPLICATES_ALLOWED to AMC\$ORDERED_BY_PRIMARY_KEY, and restarts creation of the alternate index. (All other indexes are unaffected by this change.)

If a change to AMC\$ORDER_BY_PRIMARY_KEY is not desired, set the error_limit attribute to 1. The occurrence of a trivial error (such as a duplicate key value) causes the trivial error limit to be reached and a fatal error issued. The fatal error terminates alternate index creation. No alternate indexes are created by the terminated AMP\$APPLY_KEY_DEFINITIONS procedure; however, it does perform all pending alternate key deletions.

AMP\$ABANDON_KEY_DEFINITIONS

Purpose	Discards the pending alternate key definitions or deletions.
Format	AMP\$ABANDON_KEY_DEFINITIONS (file_identifier,status);
Parameters	file_identifier : amt\$file_identifier File identifier returned by the AMP\$OPEN call for the file. status : VAR of ost\$status Status variable in which the completion status is returned.
Condition Identifiers	aae\$no_definitions_pending aae\$not_enough_permission
Remarks	A pending alternate key definition or deletion is one requested by an AMP\$CREATE_KEY_DEFINITION or AMP\$DELETE_KEY_DEFINITION call that has not yet been discarded or applied to the file. An AMP\$ABANDON_KEY_DEFINITIONS call or the closing of the file discards all pending definitions and deletions. (An AMP\$APPLY_KEY_DEFINITIONS call applies all pending definitions and deletions.)

Using Alternate Keys

An alternate key is available for use after it has been defined by an AMP\$CREATE_KEY_DEFINITION call and the definition applied by an AMP\$APPLY_KEY_DEFINITIONS call. The following sections describe how you can use an alternate key.

In general, file access calls perform the same when an alternate key is selected as when the primary key is selected. The only difference is that records are accessed through the alternate index.

Record access through the alternate index means that the logical record order is the order of the alternate key values in the alternate index. The alternate key values are stored in ascending order.

If more than one record is associated with the same alternate key value, the records are accessed in the order their primary key values occur in the key list for the alternate key value. For example, if the key list for alternate key value A contains the primary key values for records REC1 and REC3 and the key list for alternate key value B contains only the primary key value for record REC2, the records would be read sequentially: REC1, REC3, REC2.

Selecting an Alternate Key

When an indexed sequential file is opened, the system assumes that file processing is by primary key. That is, the selected key is initially the primary key. You can change the selected key by calling AMP\$SELECT_KEY. The call names the key selected.

To specify an alternate key on an AMP\$SELECT_KEY call, you specify the name of the key as it was defined when the key was created. To specify the primary key on an AMP\$SELECT_KEY call, you specify \$PRIMARY_KEY.

The key selected by an AMP\$SELECT_KEY call is used until another AMP\$SELECT_KEY call changes the selected key or until the file is closed.

File Positioning After Alternate Key Selection

When an AMP\$SELECT_KEY call selects a different key, it sets the file position to the beginning of the index for that key. (If the key specified on an AMP\$SELECT_KEY call is already the selected key, the file position is not changed.) After an alternate key is selected, all file positioning follows the logical record order represented in the alternate index.

As described earlier in this chapter, several calls are available to position an indexed sequential file. Those calls that both position the file and read and write data are described later. The following calls position the file without reading or writing data:

AMP\$START

Positions the file to access the record with the specified alternate key value.

AMP\$REWIND

Positions the file at the beginning of the alternate index. The file is positioned to access the record with the lowest alternate key value.

AMP\$SKIP

Positions the file forward or backward the specified number of records (according to the record order provided by the alternate index).

Reading Records After Alternate Key Selection

In general, the calls to read (or get) a record perform the same when an alternate key is selected as when the primary key is selected. The only difference is that records are accessed through the alternate index.

Random get calls specify the record to be read by its alternate key value. Sequential get calls access records in sorted order by alternate key value.

These calls get a record and position the file to read or write the next record. The next record is the record having the next primary key value listed in the alternate index.

AMP\$GET_KEY

Gets the first record in the key list of the specified alternate key value and positions the file to read the next record.

An AMP\$GET_KEY call specifies the alternate key value either in the location referenced by the key_location pointer or (with a NIL key_location pointer) in the working storage area. The second method is especially useful for concatenated alternate keys because the fields of the key can be assembled in the working storage area. Each key field value is stored in the working storage area at its actual position within the record.

AMP\$GET_NEXT_KEY

Gets the record at the current position in the alternate index, returns the alternate key value of the record read, and positions the file to read the next record.

The alternate key value returned is the value stored in the alternate index. If the alternate key type is `AMC$COLLATED_KEY`, the key values are stored in collated form. In collated form, each character is represented by the lowest character code having the same collating weight. For example, if lowercase letters are collated as equal to the corresponding uppercase letters (each uppercase/lowercase pair has the same collating weight), the alternate key value is stored (and later returned) using only uppercase letters.

AMP\$GET_NEXT

Gets the record at the current position in the alternate index and positions the file to read the next record.

Updating an Alternate Index

A call to put, replace, or delete a record cannot specify an alternate key value; a key value specified on a put, replace, or delete call is expected to be a primary key value even if an alternate key is currently selected. However, put, replace, and delete calls do update any alternate key indexes affected by the operation.

When a call deletes a record in the file, any alternate index entries for the record are deleted.

When a call writes a new record to the file, an entry for the record is added to the alternate indexes (unless the record is excluded from an index by sparse key control). The new record can then be read by its alternate key value.

When a call replaces an existing record in the file, the alternate index entries for the record are replaced with the appropriate entries for the new record. (The alternate key value could have changed or sparse key control could exclude the record from an alternate index.)

If an alternate index in the file was created using the default `duplicate_key_control` value `AMC$NO_DUPLICATES_ALLOWED`, a record having the same alternate key value as a record already in the file cannot be written to the file. An attempt to put or replace a duplicate record returns a trivial error and the record is not written.

Fetching Access Information After Alternate Key Selection

An AMP\$FETCH_ACCESS_INFORMATION call can return the following items of information as described in section 7. This list highlights the meaning of each item when returned immediately after a file access call that specifies an alternate key value:

`duplicate_value_inserted`

Boolean indicating whether the last put or replace call wrote a record having a duplicate alternate key value. The duplication may not be for the currently selected alternate key; it could be for any alternate key that allows duplicates.

`file_position`

Returns the current file position as described later under File Position Returned.

`primary_key`

Primary key of the record at the current file position (the next record). The AMP\$FETCH_ACCESS_INFORMATION call must specify a pointer to the location where the primary key value is to be returned.

`selected_key_name`

Name of the currently selected key. If the primary key is currently selected, \$PRIMARY_KEY is returned.

File Position Returned

At completion of each AMP\$START, AMP\$GET_KEY, or AMP\$GET_NEXT_KEY call, a value is returned in the `file_position` variable. The value returned is AMC\$EOR, AMC\$EOI, or AMC\$END_OF_KEY_LIST.

When returned by an AMP\$START call, the `file_position` values have these meanings:

`AMC$END_OF_KEY_LIST`

The alternate index is positioned at the end of a key list and at the beginning of the next key list. The next key list is for either the specified alternate key value or the next higher alternate key value if the specified value was not found.

`AMC$EOI`

The alternate index is positioned at its end because the specified alternate key value was higher than any alternate key value in the index.

When returned by an AMP\$GET_KEY call, the file_position values have these meanings:

AMC\$EOR

A record associated with the alternate key value has been returned, and if an AMP\$GET_NEXT_KEY call were issued next, it would return the next record in the key list for the alternate key value.

AMC\$END_OF_KEY_LIST

The last (or only) record associated with the alternate key value has been returned, and if an AMP\$GET_NEXT_KEY call were issued next, it would return a record with another alternate key value or the file_position AMC\$EOI.

AMC\$EOI

Same as for AMP\$START.

When returned by an AMP\$GET_NEXT_KEY call, the file_position values have these meanings:

AMC\$EOR

Same as for AMP\$GET_KEY.

AMC\$END_OF_KEY_LIST

Same as for AMP\$GET_KEY.

AMC\$EOI

No record is returned because the file is positioned at the end of the alternate index.

Retrieving Alternate Index Information

Three calls retrieve alternate index information:

`AMP$GET_NEXT_PRIMARY_KEY_LIST`

Retrieves a list of primary keys based on the current position in the index and an upperbound that you specify.

`AMP$GET_PRIMARY_KEY_COUNT`

Returns the number of primary keys based on boundaries that you specify.

`AMP$GET_KEY_DEFINITIONS`

Retrieves the definitions of existing alternate keys.

Your program could use the definitions returned by `AMP$GET_KEY_DEFINITIONS` to:

- Determine the attributes of an alternate key
- Define identical or similar alternate keys in another file

For example, when you recreate an indexed sequential file, the alternate key definitions must be redefined for the new file. By retrieving and saving the alternate key definitions of the old file, you can use the definitions when defining the alternate keys for the new file and ensure that the new alternate key definitions are identical to those of the old file.

AMP\$SELECT_KEY

Purpose	Selects the key to be used by subsequent calls.
Format	AMP\$SELECT_KEY (file_identifier,key_name,status);
Parameters	<p>file_identifier: amt\$file_identifier File identifier returned by the AMP\$OPEN call for the file.</p> <p>key_name: amt\$key_name Name of the key to be used. Specify an alternate key name or specify \$PRIMARY_KEY to switch from an alternate key back to the primary key.</p> <p>status: VAR of ost\$status Status variable in which the completion status is returned.</p>
Condition Identifiers	<p>aae\$altkey_name_not_found</p> <p>aae\$cant_select_key</p> <p>aae\$cant_select_until_applied</p> <p>aae\$no_select_on_pending_delete</p> <p>aae\$not_enough_permission</p>
Remarks	<ul style="list-style-type: none"> • The initial key selected when a file is opened is always the primary key. • The key selection remains in effect until another AMP\$SELECT_KEY call is issued or the file is closed. • AMP\$SELECT_KEY cannot select an alternate key for which a deletion request is pending (an AMP\$DELETE_KEY_DEFINITION has specified the key). If a deletion request is pending for the specified key, AMP\$SELECT_KEY returns the condition AAE\$NO_SELECT_ON_PENDING_DELETE. • When an AMP\$SELECT_KEY call changes the selected key, it positions the file at the record having the lowest key value for the selected key (that is, it rewinds the file for that key). However, if the AMP\$SELECT_KEY call does not change the selected key (the key specified on the call is already selected), it does not rewind the file (the file is left in its current position).

AMP\$GET_KEY_DEFINITIONS

Purpose Retrieves the definitions of all alternate keys in the file.

Format AMP\$GET_KEY_DEFINITIONS
(file_identifier,key_definitions,status);

Parameters **file_identifier**: amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

key_definitions: SEQ (*)

Sequence to receive the description of the alternate keys. Each definition is written in two parts: a record of type AMT\$BASIC_KEY_DEFINITION and an array of type AMT\$OPTIONAL_KEY_ATTRIBUTES containing three or more additional records. (The number of records is returned in the NUMBER_OF_OPTIONAL_ATTRIBUTES field of the AMT\$BASIC_KEY_DEFINITION record.)

status: VAR of ost\$status

Status variable in which the completion status is returned.

Condition Identifiers aae\$not_enough_permission
aae\$too_little_space

Remarks

- A successful AMP\$GET_KEY_DEFINITIONS call returns the key definitions in the sequence specified by the key_definitions parameter. The last key definition in the sequence consists of an amt\$basic_key_definition record in which the field definition_returned is FALSE; the record serves as the terminator for the sequence of key definitions.
- If the definition_returned field is TRUE in an AMT\$BASIC_KEY_DEFINITION record, the record is the first part of a key definition. The NUMBER_OF_OPTIONAL_ATTRIBUTES field in the record specifies the number of additional records returned for the key definition; the records are returns in an array of type AMT\$OPTIONAL_KEY_ATTRIBUTES.

- The SELECTOR field of an optional attribute record indicates the attribute returned in the record. The possible attributes are: key_type, duplicate_key_control, null_suppression, sparse_key_control, concatenated_key, and repeating_groups. The first three records are returned for every key definition; the subsequent records are returned only if the attribute was specified for the key definition.
- Although the attribute order in a key definition may not match the attribute order specified when the alternate key was defined, the returned definition is logically equivalent and, if used to redefine the key, results in an identical alternate key.
- All name values in an alternate key definition are returned using uppercase letters only (even if lowercase letters were used when the name was originally specified).

Example: The following CYBIL statements show how the key definition sequence returned by an AMP\$GET_KEY_DEFINITIONS call could be read. The key definition sequence is declared to be 500 words long (500 integers). If the sequence is too small, AMP\$GET_KEY_DEFINITIONS returns the condition AAE\$TOO_LITTLE_SPACE.

```

MODULE GET_DEFS_MOD;
*copyc amp$open
*copyc amp$get_key_definitions
*copyc amt$optional_key_attributes
PROCEDURE GET_ALT_KEY_DEFS;
  VAR
    lfn: [STATIC] amt$local_file_name := 'existing_is_file',
    fid: amt$file_identifier,
    status: ost$status,
    definitions_ptr : ^SEQ (*),
    definitions : SEQ(REP 500 OF integer),
    basic_definition : ^amt$basic_key_definition,
    optional_attributes : ^amt$optional_key_attributes;

    amp$open(lfn,amc$record,NIL,fid,status);

  { Statements here to check the status variable.}

    amp$get_key_definitions (fid,definitions,status);

  { Statements here to check the status variable.}

    definitions_ptr := ^definitions;
    RESET definitions_ptr;

  { Set the basic_definitions pointer to the first record.}
    NEXT basic_definition IN definitions_ptr;

  { Iterate until the definition_returned field in the }
  { basic_definition record is FALSE.}
    WHILE basic_definition^.definition_returned DO

  { Set the optional_attributes pointer to the beginning }
  { of the optional attributes array.}
    NEXT optional_attributes :
      [1 .. basic_definition^.number_of_optional_attributes]
      IN definitions_ptr;
      :
  { Use the key definition here. }
      :

  { Set the basic_definition pointer to the next key }
  { definition.}
    NEXT basic_definition IN definitions_ptr;

  WHILEND;
PROCEND GET_ALT_KEY_DEFS;
MODEND GET_DEFS_MOD

```

AMP\$GET_PRIMARY_KEY_COUNT

Purpose Returns the number of primary key values that are associated with all alternate key values in a range.

Format AMP\$GET_PRIMARY_KEY_COUNT (file_ identifier,low_key,major_low_key,low_key_ relation,high_key,major_high_key,high_key_ relation,list_count_limit,list_count,wait,status);

Parameters **file_identifier:** amt\$file_identifier

File identifier returned by the AMP\$OPEN for the file.

low_key: ^cell

Pointer to the alternate key value at which the range begins. Set to NIL if the range begins at the lowest alternate key value in the alternate index.

major_low_key: amt\$major_key_length

Specify a nonzero value to indicate that the lowerbound alternate key value is to be located by major key. The nonzero value is the number of characters beginning at the low_key location that are to be used as the major key. Specify zero to indicate that the full alternate key value is to be used.

low_key_relation: amt\$key_relation

Indicates where the count begins in relation to the lowest value in the range.

AMC\$GREATER_KEY

Exclude the primary keys associated with the low_key value from the count, that is, begin the count when an alternate key value greater than the low_key value is encountered.

AMC\$GREATER_OR_EQUAL_KEY or
AMC\$EQUAL_KEY

Include the primary keys associated with the low_key value in the count, that is, begin the count when an alternate key value greater than or equal to the low_key value is encountered.

high_key: ^cell

Pointer to the alternate key value at which the range ends. Set to NIL if the range ends at the highest alternate key value in the alternate index.

major_high_key: amt\$major_key_length

Specify a nonzero value to indicate that the upperbound alternate key value is to be located by major key. The nonzero value is the number of characters beginning at the high_key location that are to be used as the major key. Specify zero to indicate that the full alternate key value is to be used.

high_key_relation: amt\$key_relation

Indicates where the count ends in relation to the highest value in the range.

AMC\$GREATER_KEY

Include the primary keys associated with the high_key value in the count, that is, end the count when an alternate key value greater than the high_key value is encountered.

AMC\$GREATER_OR_EQUAL_KEY or
AMC\$EQUAL_KEY

Exclude the primary keys associated with the high_key value from the count, that is, end the count when an alternate key value greater than or equal to the high_key value is encountered.

list_count_limit: 0 .. amt\$key_count_limit

Maximum number of primary keys counted; the system stops counting when it reaches this value. If set to zero, all primary keys are counted.

list_count: VAR of 0 .. amt\$key_count_limit

Integer variable in which the number of primary keys in the range is returned. If zero is returned, no primary keys exist in the specified range. The value cannot exceed the list count limit.

wait: ost\$wait

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: ost\$status

Status variable in which the completion status is returned.

Condition Identifiers

aae\$high_end_not_above_low_end
aae\$not_enough_permission
aae\$not_positioned_by_altkey

Remarks

- You must call AMP\$SELECT_KEY to select the alternate key before calling AMP\$GET_PRIMARY_KEY_COUNT; otherwise, AMP\$GET_PRIMARY_KEY_COUNT returns the trivial error aae\$not_positioned_by_altkey and does not return a primary key count.
- The low_key and high_key parameters point to values that specify the lower and upper bounds, respectively, of the range of keys to be counted. The low_key_relation and high_key_relation parameters indicate whether the low_key and high_key values, respectively, are included in the range.

For example, suppose the low_key value is JONES and the high_key value is SMITH. The low_key_relation value indicates whether the primary keys associated with alternate key value JONES are included in the count. The high_key_relation value indicates whether the primary keys associated with alternate key value SMITH are included in the count.

- A major key consists of the leftmost characters of a key. The major_high_key and major_low_key parameters specify the number of characters of the specified key to use when searching for a matching key. A key is considered to match the specified key when the major key matches the first characters of the key.

For example, suppose the key at the specified low_key position is ABCDEF. If the major_low_key parameter value is 2, the major key used is AB. Therefore, the count begins at the first alternate key value beginning with a value greater than or equal to AB.

- If low_key is set to NIL, the values of major_low_key and low_key_relation are ignored. If high_key is set to NIL, the values of major_high_key and high_key_relation are ignored.
- AMP\$GET_PRIMARY_KEY_COUNT counts a single primary key value more than once if the primary key value is associated with more than one alternate key value. This is possible if the repeating groups attribute is defined for the alternate key.
- The list_count_limit value can minimize the processing required for the call. If, for example, you call AMP\$GET_PRIMARY_KEY_COUNT call to determine whether the number of primary key values is 0, 1 or more than 1, you should set the list_count_limit value to 2.

AMP\$GET_NEXT_PRIMARY_KEY_LIST

Purpose Returns a list of primary keys values corresponding to a range of alternate key values.

Format AMP\$GET_NEXT_PRIMARY_KEY_LIST (file_ identifier,high_key, major_high_key,high_key_ relation,working_storage_area, working_storage_ length,end_of_primary_key_list, transferred_byte_ count,transferred_key_count, file_position,wait, status);

Parameters **file_identifier:** amt\$file_identifier
File identifier returned by the AMP\$OPEN call for the file.

high_key: ^cell

Pointer to the alternate key value at which the range ends. Set to NIL if the range ends at the end of the alternate index.

major_high_key: amt\$major_key_length

Specify a nonzero value to indicate that the upperbound alternate key value is to be located by major key. The nonzero value is the number of characters beginning at the high_key location that are to be used as the major key. Specify zero to indicate that the full alternate key value is to be used.

high_key_relation: amt\$key_relation

Indicates where the list ends in relation to the highest alternate key value in the range.

AMC\$GREATER_KEY

Include the primary keys associated with the high_key value in the list, that is, end the list when an alternate key value greater than the high_key value is encountered.

AMC\$GREATER_OR_EQUAL_KEY or
AMC\$EQUAL_KEY

Exclude the primary keys associated with the high_key value from the list, that is, end the list when an alternate key value greater than or equal to the high_key value is encountered.

working_storage_area: ^cell

Pointer to the variable in which the primary key list is returned.

working_storage_length: amt\$working_storage_length
Length, in bytes, of the working storage area.

end_of_primary_key_list: VAR of boolean
Variable in which a boolean value is returned indicating whether the entire list of primary key values was returned to the working storage area.

TRUE

The high end of the range was reached, and the entire list of primary key values was returned to the working storage area.

FALSE

The high end of the range was not reached, and at least one more AMP\$GET_NEXT_PRIMARY_KEY_LIST call is required to get the rest of the primary key list.

transferred_byte_count: VAR of amt\$working_storage_length

Variable in which the length, in bytes, of the list of primary key values is returned.

transferred_key_count: VAR of amt\$key_count_limit

Variable in which the number of primary key values is returned.

file_position: VAR of amt\$file_position

Variable in which the file position at completion of the operation is returned.

AMC\$EOR

File is positioned within a key list.

AMC\$END_OF_KEY_LIST

File is positioned at the end of a key list.

AMC\$EOI

File is positioned at the end of the alternate index.

wait: ost\$wait

Currently, the only valid value is OSC\$WAIT. You must specify this value on the call.

status: VAR of ost\$status

Status variable in which the completion status is returned.

Condition Identifiers

aae\$high_end_below_current
 aae\$not_enough_permission
 aae\$not_positioned_by_altkey
 aae\$wsa_not_given
 aae\$wsl_too_short

Remarks

- You must call AMP\$SELECT_KEY to select the alternate key before calling AMP\$GET_NEXT_PRIMARY_KEY_LIST; otherwise, AMP\$GET_NEXT_PRIMARY_KEY_LIST returns the trivial error aae\$not_positioned_by_altkey and does not return a primary key list.

- The high_key parameter points to a value that specifies the upper bound of the range of keys to be listed. The high_key_relation parameter indicates whether the high_key value is included or excluded from the range. For example, suppose the high_key value is SMITH. The high_key_relation value indicates whether the primary keys associated with alternate key value SMITH are included in the list.

- A major key consists of the leftmost characters of a key. The major_high_key parameter specifies the number of characters of the specified key to use when searching for a matching key. A key is considered to match the specified key when the major key matches the first characters of the key.

For example, suppose the key at the specified high_key location is ABCDEF. If the major_high_key parameter value is 2, the major key used is AB. Therefore, the range ends at the first alternate key value beginning with AB.

- If high_key is set to NIL, the values of major_high_key and high_key_relation are ignored.
- A primary key value can be included more than once in the list returned by AMP\$GET_NEXT_PRIMARY_KEY_LIST. This occurs if the primary key value is associated with more than one alternate key value in the range. This is possible if the repeating groups attribute is defined for the alternate key.

Alternate Key Example

The following program illustrates the use of alternate keys. The program uses the indexed sequential file created and updated in the earlier examples in this chapter. It also uses the common procedures listed in appendix G.

The program defines the capital field as the alternate key field. It then copies the records to file ALTERNATE_KEY_OUTPUT, sorted by the alternate key.

This is a source listing of the program.

```

MODULE example_3 ;
?? left := 1, right := 110 {source line margin control} ??
?? PUSH (LIST:=OFF) ??
*copyc amp$apply_key_definitions
*copyc amp$close
*copyc amp$create_key_definition
*copyc amp$file
*copyc amp$get_next_key
*copyc amp$open
*copyc amp$put_next
*copyc amp$select_key
{ This deck contains the common procedures listed in appendix G. }
*copyc comproc
?? POP ??

```

{This module defines and then uses alternate keys for ISFILE.}

CONST

```
max_record_length = 55;
```

VAR

```
{ Declare variables for ISFILE.}
isfile: amt$local_file_name,
isfile_id: amt$file_identifier,
isfile_fpos: amt$file_position,
```

```
{Declare variables for alternate key CAPITAL_KEY.}
capital_key_name: amt$key_name := 'capital_key',
capital_key_position: amt$key_position := 41,
capital_key_length: amt$key_length := 14,
```

```
{ Declare variables for SEQFILE.}
sqfile: amt$local_file_name,
sqfile_id: amt$file_identifier,
sqfile_byte_address: amt$file_byte_address;
```

```

VAR
  wsa: string(max_record_length),
  record_length : amt$max_record_length;

{ Declare access_selections array for amp$open of ISFILE.}
VAR
  access_selections_isfile: [STATIC] array [1 .. 1] of
    amt$file_item :=
    [[amc$message_control, $amt$message_control
      [amc$trivial_errors, amc$messages, amc$statistics]]];

{ Establish the file attribute array for file_description.}
VAR
  file_description: [STATIC] array [1 .. 2] of amt$file_item :=
    [[amc$file_organization,      amc$sequential],
      [amc$max_record_length,      max_record_length]];

{ Declare access_selections array for amp$open of SEQFILE.}
VAR
  access_selections_sqfile: [STATIC] array [1 .. 1]
    of amt$file_item :=
    [[amc$file_contents,          amc$legible]];

VAR
  capital_attributes: [STATIC,READ] array [1..1]
    of amt$optional_key_attribute :=
    [[amc$duplicate_keys, amc$ordered_by_primary_key]];

PROGRAM alternate_key_phase (VAR program_status : ost$status);

  p#start_report_generation('Begin alternate keys example.');
```

{These calls specify file attributes and open files. }

```

  isfile := 'indexed';
  sqfile := 'alternate_key_output';
  amp$file (sqfile, file_description, status);
  p#inspect_status_variable;
  amp$open (isfile, amc$record, ^access_selections_isfile,
    isfile_id, status);
  p#inspect_status_variable;
  amp$open (sqfile, amc$record, ^access_selections_sqfile,
    sqfile_id, status);
  p#inspect_status_variable;
```

{These calls define and generate the alternate index. }

```

  amp$create_key_definition (isfile_id, capital_key_name,
    capital_key_position, capital_key_length,
    ^capital_attributes, status);
  p#inspect_status_variable;
  amp$apply_key_definitions (isfile_id, status);
  p#inspect_status_variable;
```

ALTERNATE KEY EXAMPLE

```
{These calls select the alternate key and read the first record. }
amp$select_key (isfile_id, capital_key_name, status);
p#inspect_status_variable;
amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
record_length, isfile_fpos, osc$wait, status);
p#inspect_status_variable ;

{ This loop copies the records in the indexed sequential }
{ file to the sequential file in the order the records }
{ are referenced in the alternate index. }
WHILE (isfile_fpos <> amc$eoi) DO
  amp$put_next (sqfile_id, ^wsa, max_record_length,
sqfile_byte_address, status);
  p#inspect_status_variable;
  wsa (1, * ) := ' ';
  amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
record_length, isfile_fpos, osc$wait, status);
  p#inspect_status_variable ;
WHILEND;

amp$close (isfile_id, status);
p#inspect_status_variable;
amp$close (sqfile_id, status);
p#inspect_status_variable ;

p#stop_report_generation('Alternate keys example complete. ');
program_status.normal := TRUE;
{ Exit with normal status. }

PROCEND alternate_key_phase;

MODEND example_3 ;
```



```

VAR
  wsa: string(max_record_length),
  record_length : amt$max_record_length;

{ Declare access_selections array for amp$open of ISFILE.}
VAR
  access_selections_isfile: [STATIC] array [1 .. 1] of
    amt$file_item :=
    [[amc$message_control, $amt$message_control
      [amc$trivial_errors, amc$messages, amc$statistics]]];

{ Establish the file attribute array for file_description.}
VAR
  file_description: [STATIC] array [1 .. 2] of amt$file_item :=
    [[amc$file_organization,    amc$sequential],
     [amc$max_record_length,    max_record_length]];

{ Declare access_selections array for amp$open of SEQFILE.}
VAR
  access_selections_sqfile: [STATIC] array [1 .. 1]
    of amt$file_item :=
    [[amc$file_contents,      amc$legible]];

VAR
  capital_attributes: [STATIC,READ] array [1..1]
    of amt$optional_key_attribute :=
    [[amc$duplicate_keys, amc$ordered_by_primary_key]];

PROGRAM alternate_key_phase (VAR program_status : ost$status);

  p#start_report_generation('Begin alternate keys example.');
```

{These calls specify file attributes and open files. }

```

  isfile := 'indexed';
  sqfile := 'alternate_key_output';
  amp$file (sqfile, file_description, status);
  p#inspect_status_variable;
  amp$open (isfile, amc$record, ^access_selections_isfile,
    isfile_id, status);
  p#inspect_status_variable;
  amp$open (sqfile, amc$record, ^access_selections_sqfile,
    sqfile_id, status);
  p#inspect_status_variable;
```

{These calls define and generate the alternate index. }

```

  amp$create_key_definition (isfile_id, capital_key_name,
    capital_key_position, capital_key_length,
    ^capital_attributes, status);
  p#inspect_status_variable;
  amp$apply_key_definitions (isfile_id, status);
  p#inspect_status_variable;
```

ALTERNATE KEY EXAMPLE

```
{These calls select the alternate key and read the first record. }
amp$select_key (isfile_id, capital_key_name, status);
p#inspect_status_variable;
amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
record_length, isfile_fpos, osc$wait, status);
p#inspect_status_variable ;

{ This loop copies the records in the indexed sequential }
{ file to the sequential file in the order the records }
{ are referenced in the alternate index. }
WHILE (isfile_fpos <> amc$eoi) DO
amp$put_next (sqfile_id, ^wsa, max_record_length,
sqfile_byte_address, status);
p#inspect_status_variable;
wsa (1, * ) := ' ';
amp$get_next_key (isfile_id, ^wsa, max_record_length, NIL,
record_length, isfile_fpos, osc$wait, status);
p#inspect_status_variable ;
WHILEND;

amp$close (isfile_id, status);
p#inspect_status_variable;
amp$close (sqfile_id, status);
p#inspect_status_variable ;

p#stop_report_generation('Alternate keys example complete. ');
program_status.normal := TRUE;
{ Exit with normal status. }

PROCEND alternate_key_phase;

MODEND example_3 ;
```

Assuming the source program is stored as deck ALTERNATE_KEYS on source library file \$USER.MY_LIBRARY, the following is a listing of the SCL commands required to expand, compile and execute the program and the output produced by the program.

```
/scu_expand_deck base=$user.my_library deck=(alternate_keys) ..
../alternate_base=($system.cybil.osf$program_interface, ..
../$system.common.psf$external_interface_source)
/cybil input=compile
/attach_file $user.indexed
/lgo
```

Begin alternate keys example.

```
-- File INDEXED : begin creating labels for alternate key
definitions.
-- File INDEXED : finished creating labels for alternate key
definitions.
-- File INDEXED : begin the data pass that collects alternate
key values.
-- File INDEXED : AMP$APPLY_KEY_DEFINITIONS has reached a file
boundary : EOI.
-- File INDEXED : data pass completed.
-- File INDEXED : begin sorting the alternate key values.
-- File INDEXED : sorting completed.
-- File INDEXED : begin building alternate key indexes into
the file.
-- File INDEXED : completed building the indexes into the file.
-- File INDEXED : AMP$GET_NEXT_KEY has reached a file
boundary : EOI.
-- File INDEXED : 0 DELETE_KEYS done since last open.
-- File INDEXED : 0 GET_KEYS done since last open.
-- File INDEXED : 48 GET_NEXT_KEYS done since
last open.
-- File INDEXED : 0 PUT_KEYS (and PUTREPs->put) since last open.
-- File INDEXED : 0 PUTREPs done since last open.
-- File INDEXED : 0 REPLACE_KEYS (and PUTREPs->replace) since
last open.
```

No error has been found by the program.

Alternate keys example complete.

ALTERNATE KEY EXAMPLE

This is a listing of the ALTERNATE_KEY_OUTPUT file written by the program.

Ivory Coast	8513000	124503	Abidjan
Algeria	19709000	919591	Algiers
Turkey	47284000	301381	Ankara
China	1053788000	3705390	Beijing
Switzerland	6300000	15941	Bern
West Germany	60948000	95976	Bonn
Belgium	9875000	11781	Brussels
Venezuela	15771000	352143	Caracas
Denmark	5157000	16629	Copenhagen
India	700734000	1269340	Delhi
Ireland	3349000	27136	Dublin
United Kingdom	55717000	94226	London
Spain	38686000	194897	Madrid
Australia	14796000	2967895	Melbourne
Mexico	70143000	761601	Mexico
USSR	269302000	8649498	Moscow
Canada	24336000	3851791	Ottawa
France	53844000	211207	Paris
Italy	57513000	116303	Rome
Sweden	8335000	173731	Stockholm
Japan	11878300	143750	Tokyo
Austria	7476000	32374	Vienna
United States	225195000	3615105	Washington
Tanzania	18744000	364898	Zanzibar

- Sequential File Organization to Sequential File Organization 11-2
- Sequential File Organization to Indexed Sequential File Organization ... 11-3
- Byte Addressable File Organization to Byte Addressable File
Organization 11-4
- Indexed Sequential File Organization to Indexed Sequential File
Organization 11-5
- Indexed Sequential File Organization to Sequential File Organization ... 11-7
- List File Copying 11-8
 - AMP\$COPY_FILE 11-9
- File Copy Example 11-11



NOS/VE offers more than one means of copying files. It provides the SCL command COPY_FILE (described in the SCL System Interface manual), the File Management Utility (FMU) (described in the SCL Advanced File Management manual), and the file interface call AMP\$COPY_FILE described in this chapter.

An AMP\$COPY_FILE call copies a file to another file. The file copied from is called the input file; the file copied to is called the output file. The file organization, block type, and record type attributes of the output file may differ from the corresponding attributes of the input file.

AMP\$COPY_FILE performs either a byte-by-byte copy or a record-by-record copy, depending on the attributes of the specified input and output files. A byte-by-byte copy does not change the physical representation of the file; the output file is an identical copy, byte by byte, of the input file. A record-by-record copy changes the physical representation of the file, although its logical content remains the same. That is, the contents of each record in the file does not change although the means of accessing each record may differ due to differing file attributes.

Table 11-1 shows the valid file organization combinations for input and output files. If an attempted copy is invalid, AMP\$COPY_FILE returns abnormal status (AME\$COPY_NOT_SUPPORTED).

Table 11-1. Valid File Organizations for AMP\$COPY_FILE

Input File	Output File		
	Sequential	Byte Addressable	Indexed Sequential
Sequential	Valid	Invalid	Valid
Byte addressable	Invalid	Valid	Invalid
Indexed sequential	Valid	Invalid	Valid

Sequential File Organization to Sequential File Organization

If both input and output files have the file organization attribute `AMC$SEQUENTIAL`, `AMP$COPY_FILE` performs either a byte-by-byte copy or a record-by-record copy.

`AMP$COPY_FILE` performs a byte-by-byte copy if all of the following conditions are met:

- Both files are mass storage files.
- Both files are opened at their beginning-of-information (BOI).
- The following file attributes of both files are identical:

```

block_type
file_access_procedure
file_contents
file_organization (AMC$SEQUENTIAL)
file_structure
record_type
    
```

Otherwise, `AMC$COPY_FILE` performs a record-by-record copy if the following conditions are met:

- The `file_structure` attributes of both files must be identical or one file has unknown file structure.
- The `file_contents` attributes of both files must be identical, or one file has unknown file contents, or the input file is legible or unknown and the output file is list. The last case is described under List File Copying.

If the `file_contents` attribute of both files is legible, the `line_number` and `statement_identifier` attributes of both files must be identical.

If the `open_position` attribute of the output file is `AMC$OPEN_AT_BOI`, `AMP$COPY_FILE` releases any data previously written on the output file.

If the `record_type` attribute of the input file is variable (V) records and the `record_type` attribute of the output file is ANSI fixed-length (F) records, the `max_record_length` attribute of the output file must be as large as the largest input record; otherwise, the copy truncates input records to the `max_record_length` value of the output file.

Sequential File Organization to Indexed Sequential File Organization

Copying from a sequential input file to an indexed sequential output file is valid if the following file attribute conditions are met:

- The file_structure attributes of both files are identical, or one file has unknown file structure.
- The file_contents attributes of both files are identical, or one file has unknown file contents.

If the open_position attribute of the output file is AMC\$OPEN_AT_BOI, AMP\$COPY_FILE releases any data previously written on the output file.

If the open_position of the output file is not AMC\$OPEN_AT_BOI, AMP\$COPY_FILE adds only those records from the input file that have unique keys. It copies records from the input file beginning at its open position. If it reads a record from the input file having a key that already belongs to a record on the output file, AMP\$COPY_FILE terminates and returns abnormal status.

When copying a sequential file to an indexed sequential file, AMP\$COPY_FILE assumes that each sequential file record has an embedded key whose location is determined by the key_length and key_position attributes of the output file.

If the output file is to have nonembedded keys, the key_length attribute of the output file specifies the length of the key in the input file record.

Copying a file from a sequential file to an indexed sequential file changes the physical representation of the file, although its logical content remains the same. That is, when you get the same record in the input file and the output file, the data is the same (assuming the key has not changed); however, if you display the input and output files, their content would appear to be different.

Byte Addressable File Organization to Byte Addressable File Organization

An AMP\$COPY_FILE call cannot copy a byte addressable file to a file having a dissimilar structure.

AMP\$COPY_FILE performs a byte-by-byte copy of a byte addressable file to another byte addressable file if the following conditions are met:

- Both files are mass storage files.
- Both files are opened at the same byte address.
- The following file attributes of both files are identical:

- block_type
 - file_access_procedure
 - file_contents
 - file_organization (byte-addressable)
 - file_structure
 - record_type

Indexed Sequential File Organization to Indexed Sequential File Organization

If both input and output files have the file organization attribute `AMC$INDEXED_SEQUENTIAL`, `AMP$COPY_FILE` performs either a byte-by-byte copy or a record-by-record copy.

`AMP$COPY_FILE` performs a byte-by-byte copy if all of the following conditions are met:

- Both files are mass storage files.
- Both files are opened at their beginning-of-information (BOI).
- The following file attributes of both files are identical:

```

block_type
collate_table_name (if key type=AMC$COLLATED_KEY)
embedded_key
file_access_procedure
file_contents
file_organization (AMC$INDEXED_SEQUENTIAL)
file_structure
key_length
key_position
key_type
max_block_length
max_record_length
min_record_length
record_type

```

Otherwise, `AMC$COPY_FILE` performs a record-by-record copy if the following conditions are met:

- The `file_structure` attributes of both files are identical, or one file has unknown file structure.
- The `file_contents` attributes of both files are identical, or one file has unknown file contents.

If the `open_position` attribute of the output file is `AMC$OPEN_AT_BOI`, `AMP$COPY_FILE` releases any data previously written on the output file.

If the `open_position` of the output file is not `AMC$OPEN_AT_BOI`, `AMP$COPY_FILE` keeps the content of the output file. It copies records from the input file; however, it only copies records with unique keys. If it reads a record from the input file having a key that already belongs to a record on the output file, `AMP$COPY_FILE` terminates and returns abnormal status.

If the input file has embedded keys and the output file has nonembedded keys, AMP\$COPY_FILE uses the key_length attribute of the output file. It uses the first key_length number of characters of the input file record as the primary key. Unless specified otherwise by a SET_FILE_ATTRIBUTES command or an AMP\$FILE call, the max_record_length attribute of the output file is assumed to be the max_record_length attribute of the input file less the value of its key_length attribute.

If the input file has nonembedded keys and the output file has embedded keys, AMP\$COPY_FILE prefixes each input record with its key when it copies the record to the output file. Unless specified otherwise by a SET_FILE_ATTRIBUTES command or AMP\$FILE call, the max_record_length attribute of the output file is assumed to be the max_record_length attribute of the input file plus its key_length attribute value.

NOTE

AMP\$COPY_FILE stores different keys for the output file than those used in the input file in the following cases:

- If the key_position attribute of the output file is not zero.
 - If the key_length attributes of the input and output files are not the same.
-

Indexed Sequential File Organization to Sequential File Organization

Copying an indexed sequential file to a sequential file is valid if the following file attribute conditions are met:

- The file_structure attributes of both files are identical, or one file has unknown file structure.
- The file_contents attributes of both files are identical, one file has unknown file contents, or the input file is legible or unknown and the output file is list. The last case is described under List File Copying.

If the file_contents attribute of both files is legible, the line_number and statement_identifier attributes must also be identical.

If the open_position attribute of the output file is AMC\$OPEN_AT_BOI, AMP\$COPY_FILE releases any data previously written on the output file.

AMP\$COPY_FILE considers the sequential file to be a file with embedded keys, the location of which in the output record is determined by the key_position and key_length attributes of the output file.

If the input file has nonembedded keys, AMP\$COPY_FILE prefixes each input record with its primary key when it copies the record to the output file. Unless specified otherwise by a SET_FILE_ATTRIBUTES command, the max_record_length attribute of the output file is set by AMP\$COPY_FILE to the max_record_length attribute of the input file plus its key_length attribute value.

NOTE

AMP\$COPY_FILE stores different keys for the output file than those used in the input file in the following cases:

- For nonembedded keys: if the key_position attribute of the output file is not zero; for embedded keys: if the key_position attributes of the input and output files are not the same.
 - If the key_length attributes of the input and output files are not the same.
-

List File Copying

If the `file_contents` attribute of the input file is legible or unknown and the `file_contents` attribute of the output file is list, `AMP$COPY_FILE` adds the following format effectors to the data:

- It adds a top-of-form format effector at the beginning of the first record of the file and at each partition boundary.
- If the `page_format` is burstable, it adds a top-of-form format effector to the first record of each page, that is, every `page_length` lines.
- If the `page_format` is nonburstable, it adds a triple-space format effector every `page_length` lines.
- It adds a single-space format effector to all other records.

AMP\$COPY_FILE

Purpose	Copies a file to another file.
Format	AMP\$COPY_FILE (input_file, output_file, status)
Parameters	<p>input_file: amt\$local_file_name; File to be copied.</p> <p>output_file: amt\$local_file_name; File to which the data is copied.</p> <p>status: VAR of ost\$status; Status record. The process identifier is AAC\$ACCESS_METHOD_ID or AMC\$ACCESS_METHOD_ID.</p>
Condition Identifiers	<p>aae\$file_at_user_record_limit aae\$file_full_no_puts_or_reps aae\$key_required ase\$not_enough_permission aae\$file_boundary_encoutered aae\$record_longer_than_wsa aae\$aam_requires_access ame\$altered_not_closed aae\$collated_key_needs_table aae\$data_pad_too_large aae\$file_reached_file_limit aae\$index_pad_too_large aae\$integer_key_gt_one_word aae\$key_length_o_or_undef aae\$max_rec_length_o_or_undef aae\$max_rec_length_too_big aae\$min_gt_max_record_length aae\$rec_too_small_for_key ame\$concurrent_tape_limit ame\$conflicting_block_types ame\$conflicting_file_addresses ame\$conflicting_file_contents ame\$conflicting_file_structures ame\$conflicting_record_types ame\$copy_not_supported ame\$copy_device_conflict ame\$empty_input_file ame\$fap_names_not_identical ame\$improper_fo_for_copy ame\$improper_fo_override ame\$improper_override_access</p>

ame\$improper_record_override
 ame\$improper_ss_block_override
 ame\$improper_us_block_override
 ame\$input_and_output_same_file
 ame\$input_file_at_eoi
 ame\$input_file_not_local
 ame\$line_numbers_unequal
 ame\$local_file_limit
 ame\$mbl_less_than_mibl
 ame\$mbl_less_than_mrl
 ame\$multiple_open_of_tape
 ame\$no_permission_for_access
 ame\$non_ANSI_blocking
 ame\$record_exceeds_mbl
 ame\$ring_validation_error
 ame\$statement_idents_unequal
 ame\$terminal_task_limit
 ame\$unable_to_load_collate_tabl
 ame\$unable_to_load_error_exit
 ame\$unable_to_load_fap
 ame\$unrecovered_read_error
 ame\$unrecovered_write_error

Remarks

- The copy begins at the position indicated by the open_ position file attribute of the input and output files.
- The copy terminates when AMP\$COPY_FILE reads the end of information (EOI) of the input file. If the input file is empty, AMP\$COPY_FILE returns abnormal status (AME\$EMPTY_INPUT_FILE).
- If the output file has not been registered in a catalog, AMP\$COPY_FILE creates the output file. Unless otherwise specified, the created output file has the same attributes as the input file (except its ring attributes).
- To specify other attributes for the created output file, you must specify the output file name on a SET_FILE_ATTRIBUTES command or AMP\$FILE call specifying the appropriate attributes.
- AMP\$COPY_FILE can perform either a byte-by-byte copy or a record-by-record copy. The copy performed depends on the file attributes of the input and output files as described in this chapter.
- For information on copying tape files, see chapter 4, Tape Management.

File Copy Example

The following CYBIL procedure copies a permanent file to a local file it creates. The caller provides the permanent file identification and password and the name to be given the local copy.

```

MODULE copy_example

?? PUSH (LISTTEXT := ON) ??
*copyc pmp$generate_unique_name
*copyc amp$attach
*copyc amp$copy_file
?? POP ??

PROCEDURE local_copy
  (permanent_file: pft$path;
   cycle_no: pft$cycle_selector;
   password: pft$password;
   local_copy_name: amt$local_file_name;
   VAR status: ost$status);

  VAR
    unique_name: ost$unique_name,
    lfn: amt$local_file_name,
    usage: pft$usage_selections,
    share: pft$share_selections,
    copy_status: ost$status;

  status.normal := true;
  usage := $pft$usage_selections[pfc$read];
  share := $pft$share_selections[pfc$read];

  /copy_operation/
  BEGIN

  { Generates the local file name for the }
  { permanent file attachment. }

  pmp$generate_unique_name (unique_name, copy_status);
  IF NOT status.NORMAL THEN
    EXIT /copy_operation/;
  IFEND;
  lfn := unique_name.value;

```

FILE COPY EXAMPLE

```
pfp$attach (lfn, permanent_file, cycle_no,  
  password, usage, share, pfc$wait, copy_status);  
IF NOT copy_status.NORMAL THEN  
  EXIT /copy_operation/;  
IFEND;  
  
amp$copy_file (lfn, local_copy_name, copy_status);  
IF NOT copy_status.NORMAL THEN  
  EXIT /copy_operation/;  
IFEND;  
  
{ Returns the attached permanent file. }  
  
amp$return (lfn, copy_status);  
END /copy_operation/;  
  
IF copy_status.NORMAL THEN  
  RETURN;  
ELSE  
  status := copy_status;  
IFEND;  
  
PROCEND local_copy;  
MODEND copy_example;
```

A

Alphabetic Character

One of the following letters:

A to Z

a to z

See Character and Alphanumeric Character.

Alphanumeric Character

An alphabetic character or a digit. See Character and Alphabetic Character.

Alternate Index

An index used to access records in an indexed sequential file by their alternate key value.

Alternate Key

A key defined for an indexed sequential file other than the primary key. See also Primary Key.

B

Beginning-of-Information (BOI)

The point at which file data begins. The byte address at the beginning-of-information is always zero.

Bit

A binary digit. A bit has the value 0 or 1. See Byte.

Block

A logical or physical grouping of data. A disk block or tape block is a physical unit of data written to the storage medium in a single operation. A block within an indexed sequential file is a logical unit of data. See Data Block and Index Block.

Byte

A group of bits. For NOS/VE, a byte is 8 bits. An ASCII character code uses the rightmost 7 bits of 1 byte.

Byte Addressable File Organization

A file organization in which records are accessed by their byte address.

C

Catalog

A directory of files and catalogs maintained by the system for a user. The catalog \$LOCAL contains only file entries (no catalog entries).

Also, 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 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.

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.

Close Operation

A set of terminating operations performed on a file when input and output operations are complete. All files must be closed before the task terminates.

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.

Collation Table

A data structure that orders a set of characters. The character order is used when sorting keys in an indexed sequential file.

Concatenated Key

An alternate key comprising two or more noncontiguous fields within a record.

Creation Run

All processing of a file, from open to close, the first time the file is written.

D

Data Block

A block in an indexed sequential file in which data records are stored. Contrast with Index Block.

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.

Device Class

The type of device with which a file definition is associated. When the file is opened, it is assigned to a device in the device class.

E

Embedded Key

Primary key that is located in the record.

End-of-Information Byte Address

The address of the byte following the last valid byte of data in a file. It is also the number of valid data bytes in the file.

End-of-Information (EOI)

The point at which data in the file ends.

Exception Condition

A situation that, when detected by a procedure caller, indicates an abnormal completion of the called procedure.

Execution Ring

The level of hardware protection assigned to a procedure while it is executing.

F

Field

A subdivision of a record that is referenced by name. For example, the field NORMAL in a record named OLD STATUS is referenced as follows:

OLD STATUS.NORMAL

File

A collection of information referenced by a name.

FAP

File Access Procedure

File Attribute

A characteristic of a file. Each file has a set of attributes that completely define file structure and processing limitations.

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

See File.

Flushing

The process of writing to disk parts of a file whose images in real memory have been altered since the file was last written to disk. Flushing does not alter the logical status or position of a file.

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 file organization in which records are accessed using a primary key. Indexed sequential files contain data blocks and index blocks.

Instance of Open

A particular opening of a file as distinguished from all other openings of the file. The system assigns each instance of open a unique file identifier. Closing the file ends the instance of open.

Integer Key

A signed binary key used with an indexed sequential file. Integer keys are sorted by arithmetic value. It can be from 1 - 8 bytes in length.

J

Job

A set of tasks executed for a user name. NOS/VE accepts interactive and batch jobs.

Job Library List

Object libraries included in the program library list for each program executed in the job.

K

Key

A string of bytes used to access a record in an indexed sequential file. See Primary Key and Major Key.

Key List

The primary key values associated with an alternate key value within an alternate index.

L

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.

M

Major Key

A high-order portion of a primary key in an indexed sequential file. A major key is used to position a file to a specific record.

Mass Storage

Disk storage that allows random file access and permanent file storage.

N

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.

Null Suppression

Alternate key attribute indicating that records with null alternate key values are not included in the alternate index.

O

Open Operation

A set of preparatory operations performed on a file before file input and output can occur.

P

Padding

Space deliberately left unused. Fixed-length records are padded if the data provided for the record is shorter than the record length. Within an indexed sequential file, blocks are padded during file creation to allow easy addition or expansion during later file updates.

Page

An allocatable unit of real memory.

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.

Pointer

The virtual address of a value.

Primary Key

The key of a record in an indexed sequential file. The primary key 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.

Program Library List

A list of object libraries searched for modules during the loading of a program.

R

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 disk files. Contrast with Sequential Access.

Record

A set of related data treated as a unit; also, a CYBIL data structure.

Repeating Groups

An alternate key attribute indicating that each data record contains a sequence of one or more alternate key values.

Rewind

An operation that positions a file at the beginning-of-information.

Ring

The level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings.

See Execution Ring.

Ring Attribute

A file attribute whose value consists of three ring numbers, referred to as r1, r2, and r3. The ring numbers define the four ring brackets for the file as follows:

Read bracket is 1 through r2.

Write bracket is 1 through r1.

Execute bracket is r1 through r2.

Call bracket is r2+1 through r3.

S

SCL Procedure

A sequence of SCL commands executed when the procedure name is entered. It can be stored as a module on an object library.

Segment

One or more pages assigned to a file. The segment has the ring attributes of the file.

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.

Sparse Key

A one-character flag that indicates whether an alternate key value is included in an alternate index.

T

Tapemark

The physical delimiter of data on a tape.

Task

The instance of execution of a program.

Terminal Attribute

A characteristic of an interactive terminal class.

Terminal Class

Interactive terminal type; the system associates a set of default terminal attributes with the terminal type.

U

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.

W

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.



ASCII Character Set

B

Table B-1 lists the ASCII character set used by the NOS/VE system.

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.4-1977). NOS/VE represents each 7-bit ASCII code in an 8-bit byte. The 7 bits are right-justified in each byte. For ASCII characters, the leftmost bit is always zero.

Table B-1. ASCII Character Set

ASCII Code			Graphic or Mnemonic	Name or Meaning
Decimal	Hexadecimal	Octal		
000	00	000	NUL	Null
001	01	001	SOH	Start of heading
002	02	002	STX	Start of text
003	03	003	ETX	End of text
004	04	004	EOT	End of transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal tabulation
010	0A	012	LF	Line feed
011	0B	013	VT	Vertical tabulation
012	0C	014	FF	Form feed
013	0D	015	CR	Carriage return
014	0E	016	SO	Shift out
015	0F	017	SI	Shift in
016	10	020	DLE	Data link escape
017	11	021	DC1	Device control 1
018	12	022	DC2	Device control 2
019	13	023	DC3	Device control 3
020	14	024	DC4	Device control 4
021	15	025	NAK	Negative acknowledge
022	16	026	SYN	Synchronous idle
023	17	027	ETB	End of transmission block
024	18	030	CAN	Cancel
025	19	031	EM	End of medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File separator
029	1D	035	GS	Group separator
030	1E	036	RS	Record separator
031	1F	037	US	Unit separator
032	20	040	SP	Space
033	21	041	!	Exclamation point
034	22	042	"	Quotation marks
035	23	043	#	Number sign
036	24	044	\$	Dollar sign
037	25	045	%	Percent sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(Opening parenthesis
041	29	051)	Closing parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant

(Continued)

Table B-1. ASCII Character Set (Continued)

ASCII Code			Graphic or Mnemonic	Name or Meaning
Decimal	Hexadecimal	Octal		
048	30	060	0	Zero
049	31	061	1	One
050	32	062	2	Two
051	33	063	3	Three
052	34	064	4	Four
053	35	065	5	Five
054	36	066	6	Six
055	37	067	7	Seven
056	38	070	8	Eight
057	39	071	9	Nine
058	3A	072	:	Colon
059	3B	073	;	Semicolon
060	3C	074	<	Less than
061	3D	075	=	Equals
062	3E	076	>	Greater than
063	3F	077	?	Question mark
064	40	100	@	Commercial at
065	41	101	A	Uppercase A
066	42	102	B	Uppercase B
067	43	103	C	Uppercase C
068	44	104	D	Uppercase D
069	45	105	E	Uppercase E
070	46	106	F	Uppercase F
071	47	107	G	Uppercase G
072	48	110	H	Uppercase H
073	49	111	I	Uppercase I
074	4A	112	J	Uppercase J
075	4B	113	K	Uppercase K
076	4C	114	L	Uppercase L
077	4D	115	M	Uppercase M
078	4E	116	N	Uppercase N
079	4F	117	O	Uppercase O
080	50	120	P	Uppercase P
081	51	121	Q	Uppercase Q
082	52	122	R	Uppercase R
083	53	123	S	Uppercase S
084	54	124	T	Uppercase T
085	55	125	U	Uppercase U
086	56	126	V	Uppercase V
087	57	127	W	Uppercase W
088	58	130	X	Uppercase X
089	59	131	Y	Uppercase Y
090	5A	132	Z	Uppercase Z
091	5B	133	[Opening bracket

(Continued)

CHARACTER SET

Table B-1. ASCII Character Set (Continued)

Decimal	ASCII Code		Graphic or Mnemonic	Name or Meaning
	Hexadecimal	Octal		
092	5C	134	\	Reverse slant
093	5D	135]	Closing bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening brace
124	7C	174		Vertical line
125	7D	175	}	Closing brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Constant and Type Declarations C

This appendix lists the constant and type declarations used by the procedures described in this manual. In general, the declarations are listed in alphabetical order by identifier name. However, the numeric order of ordinal constants is maintained. Also, the FAP call block declarations are listed separately following the AM type declarations.

AM

Constants

```
aac$access_method_ID = 'AA';
amc$access_method_id = 'AM';
amc$apl = 'APL',
amc$assembler = 'ASSEMBLER';
amc$basic = 'BASIC';
amc$cobol = 'COBOL';
amc$cybil = 'CYBIL';
amc$data = 'DATA';
amc$debugger = 'DEBUGGER';
amc$file_byte_limit = 4398046511103 { 2**42 - 1 }
{ bytes } ;
amc$fortran = 'FORTRAN';
amc$legible = 'LEGIBLE';
amc$library = 'LIBRARY';
amc$list = 'LIST';
amc$max_length = 2048 { bytes } ;

amc$max_attribute = 511 { 01ff(16) } ;
amc$max_block_number = 0ffffff(16);
amc$max_blocks_per_file = amc$file_byte_limit DIV
    amc$mau_length;
amc$max_buffer_length = 16777215 { 2**24 - 1 bytes } ;
amc$max_ecc_program_action = 161999;
amc$max_ecc_validation = 160999;
amc$max_error_count = 0ffff(16);
amc$max_fap_layers = 15;
amc$max_file_id_ordinal = 4095;
amc$max_home_blocks = amc$file_byte_limit;
amc$max_index_level = 15;
amc$max_info = 01ff(16);
amc$max_key_length = 255,
amc$max_key_position = 0ffff(16),
amc$max_label_length = osc$maximum_offset;
```

CONSTANT AND TYPE DECLARATIONS

```
amc$max_line_number = 6;
amc$max_lines_per_inch = 12,
amc$max_operation = 01ff(16);
amc$max_page_width = 65535;
amc$max_record_header = 16;
amc$max_records_per_block = 0ffff(16);
amc$max_path_name_size = 256;
amc$max_statement_id_length = 17;
amc$max_tape_mark_count = 40000;
amc$max_user_info = 32;
amc$max_vol_number = 64;

amc$maximum_block = 16777216 { 2**24 bytes } ;
amc$maximum_record = amc$file_byte_limit;

amc$min_ecc_program_action = 161000;
amc$min_ecc_validation = 160000;

amc$object = 'OBJECT';
amc$pascal = 'PASCAL';
amc$pli = 'PLI';
amc$ppu_assembler = 'PPU_ASSEMBLER';
amc$scl = 'SCL';
amc$scu = 'SCU';
amc$unknown_contents = 'UNKNOWN';
amc$unknown_processor = 'UNKNOWN';
amc$unknown_structure = 'UNKNOWN';
```

Ordinals

☐

{Codes 1..100 are reserved for operations which are}
{not passed to file_access_procedures.}

☐

```

amc$access_method_req = 1,
amc$add_to_file_description_req = 3,
amc$allocate_req = 5,
amc$change_file_attributes_cmd = 6,
amc$compare_file_cmd = 7,
amc$copy_file_cmd = 8,
amc$copy_file_req = 9,
amc$copy_partitions_req = 10,
amc$copy_records_req = 11,
amc$copy_partial_records_req = 12,
amc$detach_file_cmd = 17,
amc$display_file_attributes_cmd = 18,
amc$display_file_cmd = 19,
amc$evict_req = 20,
amc$fetch_fap_pointer_req = 22,
amc$file_req = 24,
amc$get_file_attributes_req = 30,
amc$label_req = 50,
amc$override_file_attributes = 60,
amc$rename_req = 72,
amc$return_req = 74,
amc$rewind_files_cmd = 75,
amc$set_local_name_abnormal_req = 76,
amc$set_file_attributes_cmd = 77,
amc$set_file_inst_abnormal_req = 78,
amc$skip_tape_marks_cmd = 81,
amc$skip_tape_marks_req = 82,
amc$store_fap_pointer_req = 84,
amc$validate_caller_privilege = 95,

```

☐

CONSTANT AND TYPE DECLARATIONS

```
{ Codes amc$fap_op_start..(amc$last_access_start-1) are }
{ reserved for operations which are passed to }
{ file_access_procedures but which are not recorded in }
{ last_access_operation status. }
{}
    amc$fap_op_start = 101,
    amc$fetch_access_information_rq = 101,
{}
{ Codes amc$last_access_start..amc$max_operation are }
{ reserved for operations which are passed to }
{ file_access_procedures. }
{}
    amc$last_access_start = 105,
    amc$check_buffer_req = 110,
    amc$check_record_req = 111,
    amc$close_req = 112,
    amc$close_volume_req = 113,
    amc$delete_req = 114,
    amc$delete_direct_req = 115,
    amc$delete_key_req = 116,
    amc$fetch_req = 117,
    amc$flush_req = 118,
    amc$get_direct_req = 119,
    amc$get_key_req = 120,
    amc$get_label_req = 121,
    amc$get_next_req = 122,
    amc$get_next_key_req = 123,
    amc$get_partial_req = 124,
    amc$get_segment_pointer_req = 126,
    amc$lock_file_req = 127,
    amc$lock_file = 127,
    amc$open_req = 128,
    amc$pack_block_req = 129,
    amc$pack_record_req = 130,
    amc$put_direct_req = 131,
    amc$put_key_req = 132,
    amc$put_label_req = 133,
    amc$put_next_req = 134,
    amc$put_partial_req = 135,
    amc$putrep_req = 137,
    amc$read_req = 138,
```

```
amc$read_direct_req = 139,  
amc$read_direct_skip_req = 140,  
amc$read_skip_req = 141,  
amc$replace_req = 142,  
amc$replace_direct_req = 143,  
amc$replace_key_req = 144,  
amc$rewind_req = 145,  
amc$rewind_volume_req = 146,  
amc$seek_direct_req = 147,  
amc$set_segment_eoi_req = 148,  
amc$set_segment_position_req = 149,  
amc$skip_req = 150,  
amc$start_req = 151,  
amc$store_req = 152,  
amc$unlock_file_req = 153,  
amc$unlock_file = 153,  
amc$unpack_block_req = 154,  
amc$unpack_record_req = 155,  
amc$write_req = 156,  
amc$write_direct_req = 157,  
amc$write_end_partition_req = 158,  
amc$write_tape_mark_req = 159,  
ifc$fetch_terminal_req = 160,  
ifc$store_terminal_req = 161,  
amc$abandon_key_definitions = 162,  
amc$abort_file_parcel = 163,  
amc$apply_key_definitions = 164,  
amc$begin_file_parcel = 165,  
amc$check_nowait_request = 166,  
amc$commit_file_parcel = 167,  
amc$create_key_definition = 168,  
amc$create_nested_file = 169,  
amc$delete_key_definition = 170,  
amc$delete_nested_file = 171,  
amc$find_record_space = 172,  
amc$get_key_definitions = 173,  
amc$get_lock_keyed_record = 174,  
amc$get_lock_next_keyed_record = 175,  
amc$get_nested_file_definitions = 176,  
amc$get_next_primary_key_list = 177,  
amc$get_primary_key_count = 178,  
amc$get_space_used_for_key = 179,  
amc$lock_key = 180,  
amc$select_key = 181,  
amc$select_nested_file = 182,  
amc$separate_key_groups = 183,  
amc$unlock_key = 184,
```

☐

CONSTANT AND TYPE DECLARATIONS

```
    amc$block_number = 1,  
    amc$current_byte_address = 2,  
    amc$eoi_byte_address = 3,  
    amc$error_count = 4 { Supported only for }  
{ indexed_sequential files },  
    amc$error_status = 5,  
    amc$file_position = 6,  
    amc$last_access_operation = 7,  
    amc$last_op_status = 8,  
    amc$levels_of_indexing = 9 { Supported only for }  
{ indexed_sequential files },  
    amc$previous_record_address = 10,  
    amc$previous_record_length = 11,  
    amc$residual_skip_count = 12,  
    amc$volume_position = 13,  
    amc$volume_number = 14,  
  
{  
    amc$access_level = 1,  
    amc$access_mode = 2,  
    amc$application_info = 3,  
    amc$average_record_length = 4,  
    amc$block_type = 5,  
    amc$character_conversion = 6,  
    amc$clear_space = 7,  
    amc$collate_table = 8,  
    amc$collate_table_name = 9,  
    amc$data_padding = 12,  
    amc$embedded_key = 13,  
    amc$error_exit_name = 14,  
    amc$error_exit_procedure = 15,  
    amc$error_limit = 16,  
    amc$error_options = 17,  
    amc$estimated_record_count = 18,  
    amc$file_access_procedure = 19,  
    amc$file_contents = 20,  
    amc$file_length = 21,  
    amc$file_limit = 22,  
    amc$file_organization = 24,  
    amc$file_processor = 25,  
    amc$file_structure = 26,  
    amc$forced_write = 27,  
    amc$global_access_mode = 28,  
    amc$global_file_address = 29,  
    amc$global_file_position = 30,  
    amc$global_file_name = 31,
```

```
amc$global_share_mode = 32,  
amc$index_levels = 33,  
amc$index_padding = 34,  
amc$internal_code = 35,  
amc$key_length = 36,  
amc$key_position = 37,  
amc$key_type = 38,  
amc$label_exit_name = 39,  
amc$label_exit_procedure = 40,  
amc$label_options = 41,  
amc$label_type = 42,  
amc$line_number = 44,  
amc$max_block_length = 45,  
amc$max_record_length = 46,  
amc$message_control = 47,  
amc$min_block_length = 48,  
amc$min_record_length = 49,  
amc$null_attribute = 50,  
amc$open_position = 51,  
amc$padding_character = 52,  
amc$page_format = 53,  
amc$page_length = 54,  
amc$page_width = 55,  
amc$permanent_file = 56,  
amc$preset_value = 57,  
amc$record_limit = 59,  
amc$record_type = 60,  
amc$records_per_block = 61,  
amc$return_option = 62,  
amc$ring_attributes = 63,  
amc$statement_identifier = 64,  
amc$user_info = 66,  
amc$vertical_print_density = 67,  
amc$compression_procedure_name = 68,  
amc$dynamic_home_block_space = 69,  
amc$hashing_procedure_name = 70,  
amc$initial_home_block_count = 71,  
amc$loading_factor = 72,  
amc$lock_expiration_time = 73,  
amc$logging_options = 74,  
amc$log_residence = 75,
```

□

```

    amc$concatenated_key_portion = 100,
    amc$duplicate_keys = 101,
    amc$group_name = 102,
    amc$null_suppression = 103,
    amc$repeating_group = 104,
    amc$sparse_keys = 105,
    amc$variable_length_key = 106,

```

Types

```

amt$access_info = record
  item_returned {output} : boolean,
  case key { input } : amt$access_info_keys of
{ output }
  = amc$block_number =
    block_number: amt$block_number,
  = amc$current_byte_address =
    current_byte_address: amt$file_byte_address,
  = amc$eoi_byte_address =
    eoi_byte_address: amt$file_byte_address,
  = amc$error_count =
    error_count: amt$error_count,
  = amc$error_status =
    error_status: ost$status_condition,
  = amc$file_position =
    file_position: amt$file_position,
  = amc$last_access_operation =
    last_access_operation:
      amt$last_access_operation,
  = amc$last_op_status =
    last_op_status: amt$last_op_status,
  = amc$levels_of_indexing =
    levels_of_indexing: amt$index_levels,
  = amc$previous_record_address =
    previous_record_address: amt$file_byte_address,
  = amc$previous_record_length =
    previous_record_length: amt$max_record_length,
  = amc$residual_skip_count =
    residual_skip_count: amt$residual_skip_count,
  = amc$volume_position =
    volume_position: amt$volume_position,
  = amc$volume_number =
    volume_number: amt$volume_number,
  casend,
recend;

```



```

amt$access_info_keys = 1 .. amc$max_info;

amt$access_information = array [1 .. * ] of
  amt$access_info;

amt$access_level = (amc$physical, amc$record,
  amc$segment);

amt$access_selection = amt$file_item;

amt$add_to_attributes = array [1 .. * ] of
  amt$add_to_item;

amt$add_to_item = record
  case key { input } : amt$file_attribute_keys of
  { input }
    = amc$character_conversion =
      character_conversion: boolean,
    = amc$file_contents =
      file_contents: amt$file_contents,
    = amc$file_limit =
      file_limit: amt$file_limit,
    = amc$file_processor =
      file_processor: amt$file_processor,
    = amc$file_structure =
      file_structure: amt$file_structure,
    = amc$forced_write =
      forced_write: amt$forced_write,
    = amc$internal_code =
      internal_code: amt$internal_code,
    = amc$line_number =
      line_number: amt$line_number,
    = amc$max_block_length =
      max_block_length: amt$max_block_length,
    = amc$max_record_length =
      max_record_length: amt$max_record_length,
    = amc$min_block_length =
      min_block_length: amt$min_block_length,
    = amc$min_record_length =
      min_record_length: amt$min_record_length,
    = amc>null_attribute =
      ,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$padding_character =  
padding_character: amt$padding_character,  
= amc$page_format =  
page_format: amt$page_format,  
= amc$page_length =  
page_length: amt$page_length,  
= amc$page_width =  
page_width: amt$page_width,  
= amc$record_type =  
record_type: amt$record_type,  
= amc$statement_identifier =  
statement_identifier: amt$statement_identifier,  
= amc$user_info =  
user_info: amt$user_info,  
= amc$vertical_print_density =  
vertical_print_density:  
    amt$vertical_print_density,  
= amc$average_record_length =  
average_record_length:  
    amt$average_record_length,  
= amc$collate_table =  
collate_table: ^amt$collate_table,  
= amc$data_padding =  
data_padding: amt$data_padding,  
= amc$embedded_key =  
embedded_key: boolean,  
= amc$estimated_record_count =  
estimated_record_count:  
    amt$estimated_record_count,  
= amc$index_levels =  
index_levels: amt$index_levels,  
= amc$index_padding =  
index_padding: amt$index_padding,  
= amc$key_length =  
key_length: amt$key_length,  
= amc$key_position =  
key_position: amt$key_position,  
= amc$key_type =  
key_type: amt$key_type,  
= amc$record_limit =  
record_limit: amt$record_limit,  
= amc$records_per_block =  
records_per_block: amt$records_per_block,  
casend,  
recend;
```

```

amt$attribute_source = (amc$undefined_attribute,
    amc$local_file_information,
    amc$change_file_attributes_req, amc$open file__request,
    amc$file_reference, amc$file_command,
    amc$file_request, amc$add_to_file_desc_request,
    amc$access_method_default, amc$put_instance_
    attributes_req;

amt$average_record_length = 1 .. amc$maximum_record;

amt$basic_key_definition = record
    case definition_returned: boolean of
    = TRUE =
        key_name: amt$key_name,
        key_position: amt$key_position,
        key_length: amt$key_length,
        number_of_optional_attributes: amt$max_optional_attributes,
        casend,
    recend,

amt$begin_file_parcel = record
    general_commit: amt$general_commit,
    recend;

    amt$block_header_type = (amc$stapemark_block,
        amc$data_block);

amt$block_number = 1 .. amc$max_block_number;

amt$block_status = (amc$no_error,
    amc$recovered_error, amc$unrecovered_error);

amt$block_type = (amc$system_specified,
    amc$user_specified);

amt$buffer_area = ^SEQ ( * );

amt$buffer_length = amc$mau_length ..
    amc$max_buffer_length;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$call_block = record
  case operation: amt$fap_operation of
  = amc$abandon_key_definitions =
  /
  = amc$abort_file_parcel =
  /
  = amc$apply_key_definitions =
  /
  = amc$begin_file_parcel =
  begin_file_parcel: amt$begin_file_parcel,
  = amc$check_buffer_req =
  check_buffer: amt$check_buffer_req,
  = amc$check_nowait_request =
  check_nowait_request: amt$check_nowait_request,
  = amc$check_record_req =
  check_record: amt$check_record_req,
  = amc$close_req =
  /
  = amc$close_volume_req =
  /
  = amc$commit_file_parcel =
  commit_file_parcel: amt$commit_file_parcel,
  = amc$create_key_definition =
  create_key_definition: amt$create_key_definition,
  = amc$create_nested_file =
  create_nested_file: amt$create_nested_file,
  = amc$delete_req =
  /
  = amc$delete_direct_req =
  deld: amt$delete_direct_req,
  = amc$delete_key_req =
  delk: amt$delete_key_req,
  = amc$delete_key_definition =
  delete_key_definition: amt$delete_key_definition,
  = amc$delete_nested_file =
  delete_nested_file: amt$delete_nested_file,
  = amc$fetch_req =
  fetch: amt$fetch_req,
  = amc$fetch_access_information_rq =
  fai: amt$fetch_access_information_rq,
  = amc$find= record_space =
  find_record_space: amt$find_record_space,
  = amc$flush_req =
  flush: amt$flush_req,
```

```

= amc$get_direct_req =
  getd: amt$get_direct_req,
= amc$get_key_req =
  getk: amt$get_key_req,
= amc$get_key_definitions =
  get_key_definitions: amt$get_key_definitions,
= amc$get_label_req =
  getl: amt$get_label_req,
= amc$get_lock_keyed_record =
  get_lock_keyed_record: amt$get_lock_keyed_record,
= amc$get_lock_next_keyed_record =
  get_lock_next_keyed_record: amt$get_lock_next_keyed_record,
= amc$get_nested_file_definitions =
  get_nested_file_definitions: amt$get_nested_file_definitions,
= amc$get_next_req =
  getn: amt$get_next_req,
= amc$get_next_key_req =
  getnk: amt$get_next_key_req,
= amc$get_next_primary_key_list =
  get_next_primary_key_list: amt$get_next_primary_key_list,
= amc$get_partial_req =
  getp: amt$get_partial_req,
= amc$get_primary_key_count =
  get_primary_key_count: amt$get_primary_key_count,
= amc$get_segment_pointer_req =
  getsegp: amt$get_segment_pointer_req,
= amc$get_space_used_for_key =
  get_space_used_for_key: amt$get_space_used_for_key,
= amc$lock_file_req =
  lock: amt$lock_file_req,
= amc$lock_key =
  lock_key: amt$lock_key,
= amc$open_req =
  open: amt$open_req,
= amc$pack_block_req =
  packb: amt$pack_block_req,
= amc$pack_record_req =
  packr: amt$pack_record_req,
= amc$put_direct_req =
  putd: amt$put_direct_req,
= amc$put_key_req =
  putk: amt$put_key_req,
= amc$put_label_req =
  putl: amt$put_label_req,
= amc$put_next_req =
  putn: amt$put_next_req,
= amc$put_partial_req =
  putp: amt$put_partial_req,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$putrep_req =  
  putrep: amt$putrep_req,  
= amc$read_req =  
  rsq: amt$read_req,  
= amc$read_direct_req =  
  rba: amt$read_direct_req,  
= amc$read_direct_skip_req =  
  rbskp: amt$read_direct_skip_req,  
= amc$read_skip_req =  
  rsqskp: amt$read_skip_req,  
= amc$replace_req =  
  replace: amt$replace_req,  
= amc$replace_direct_req =  
  repld: amt$replace_direct_req,  
= amc$replace_key_req =  
  repk: amt$replace_key_req,  
= amc$rewind_req =  
  rewind: amt$rewind_req,  
= amc$rewind_volume_req =  
  rewwol: amt$rewind_volume_req,  
= amc$seek_direct_req =  
  seekd: amt$seek_direct_req,  
= amc$select_key =  
  select_key: amt$select_key,  
= amc$select_nested_file =  
  select_nested_file: amt$select_nested_file,  
= amc$separate_key_groups =  
  separate_key_groups: amt$separate_key_groups,  
= amc$set_segment_eoi_req =  
  segeoi: amt$set_segment_eoi_req,  
= amc$set_segment_position_req =  
  segpos: amt$set_segment_position_req,  
= amc$skip_req =  
  skp: amt$skip_req,  
= amc$start_req =  
  start: amt$start_req,  
= amc$store_req =  
  store: amt$store_req,  
= amc$unlock_file_req =
```

```

= amc$unlock_key =
  unlock_key: amt$unlock_key,
= amc$unpack_block_req =
  unpackb: amt$unpack_block_req,
= amc$unpack_record_req =
  unpackr: amt$unpack_record_req,
= amc$write_req =
  wsq: amt$write_req,
= amc$write_direct_req =
  wba: amt$write_direct_req,
= amc$write_end_partition_req =
  ,
= amc$write_tape_mark_req =
  ,
= ifc$fetch_terminal_req =
  fetch_terminal: ift$fetch_terminal_req,
= ifc$store_terminal_req =
  store_terminal: ift$store_terminal_req,
casend,
recend;

amt$check_nowait_request = record
  request_complete: ^boolean,
  returned_parameters: ^amt$nowait_var_parameters,
  request_status: ^ost$status,
  wait: ost$wait,
recend;

amt$collate_table = array [char] of
  amt$collation_value;

amt$collation_value = 0 .. 255;

amt$commit_file_parcel = record
  phase: amt$commit_phase,
recend;

amt$commit_phase = (amc$simple_commit, amc$tentative_commit,
  amc$permanent_commit);

amt$compression_effect = (amc$compress, amc$decompress);

```

CONSTANT AND TYPE DECLARATIONS

```
amt$compression_procedure = ^procedure (effect: amt$compression_effect;  
  input_working_storage_area: ^cell;  
  input_working_storage_length: amt$max_record_length;  
  output_working_storage_area: ^cell;  
  key_position: amt$key_position;  
  key_length: amt$key_length;  
  VAR output_working_storage_length: amt$max_record_length;  
  VAR record_left_uncompressed: boolean;  
  VAR status: ost$status);  
  
amt$compression_procedure_name = amt$entry_point_reference;  
  
amt$create_key_definition = record  
  key_name: amt$key_name,  
  key_position: amt$key_position,  
  key_length: amt$key_length,  
  optional_attributes: ^amt$optional_key_attributes,  
recend;  
  
amt$create_nested_file = record  
  definition: ^amt$nested_file_definition,  
recend;  
  
amt$creation_date = 1 .. 99999 { yyddd, defaults }  
  { to current date } ;  
  
amt$data_block_count = 1 .. amc$max_blocks_per_file;  
  
amt$data_padding = 0 .. 99 { expressed as a }  
  { percentage } ;  
  
amt$delete_key_definition = record  
  key_name: amt$key_name,  
recend;  
  
amt$delete_nested_file = record  
  nested_file_name: amt$nested_file_name,  
recend;  
  
amt$duplicate_key_control = (amc$no_duplicates_allowed,  
  amc$first_in_first_out, amc$ordered_by_primary_key);  
  
amt$duplicate_value_inserted = boolean;  
  
amt$dynamic_home_block_space = boolean;  
  
amt$error_count = 0 .. amc$max_error_count;
```



```

amt$entry_point_reference = record
  name: pmt$program_name,
  object_library: amt$path_name,
recend;

amt$error_exit_procedure = ^procedure
  (file_identifier: amt$file_identifier;
  VAR status: ost$status);

amt$error_limit = 0 .. 0ffff(16);

amt$error_options = (amc$terminate_file,
  amc$drop_block, amc$accept_record);

amt$estimated_record_count = integer;

amt$expiration_date = 1 .. 99999 { yyddd, defaults }
{ to creation date plus one year };

amt$fap_layer_number = 0 .. amc$max_fap_layers;

amt$fap_operation = amc$fap_op_start ..
  amc$max_operation;

amt$fap_pointer = ^procedure
  (file_identifier: amt$file_identifier;
  call_block: amt$call_block;
  layer_number: amt$fap_layer_number;
  VAR status: ost$status);

amt$fetch_attributes = array [1 .. * ] of
  amt$fetch_item;

amt$fetch_item = record
  source { output } : amt$attribute_source,
  case key { input } : amt$file_attribute_keys of
{ output }
  = amc$access_level =
    access_level: amt$access_level,
  = amc$access_mode =
    access_mode: pft$usage_selections,
  = amc$application_info =
    application_info: pft$application_info,
  = amc$block_type =
    block_type: amt$block_type,
  = amc$character_conversion =
    character_conversion: boolean,
  = amc$clear_space =
    clear_space: ost$clear_file_space,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$error_exit_name =  
  error_exit_name: pmt$program_name,  
= amc$error_exit_procedure =  
  error_exit_procedure: amt$error_exit_procedure,  
= amc$error_options =  
  error_options: amt$error_options,  
= amc$file_access_procedure =  
  file_access_procedure: pmt$program_name,  
= amc$file_contents =  
  file_contents: amt$file_contents,  
= amc$file_limit =  
  file_limit: amt$file_limit,  
= amc$file_organization =  
  file_organization: amt$file_organization,  
= amc$file_processor =  
  file_processor: amt$file_processor,  
= amc$file_structure =  
  file_structure: amt$file_structure,  
= amc$forced_write =  
  forced_write: amt$forced_write,  
= amc$global_access_mode =  
  global_access_mode: pft$usage_selections,  
= amc$global_file_address =  
  global_file_address: amt$file_byte_address,  
= amc$global_file_name =  
  global_file_name: ost$binary_unique_name,  
= amc$global_file_position =  
  global_file_position: amt$global_file_position,  
= amc$global_share_mode =  
  global_share_mode: pft$share_selections,  
= amc$internal_code =  
  internal_code: amt$internal_code,  
= amc$label_exit_name =  
  label_exit_name: pmt$program_name,  
= amc$label_exit_procedure =  
  label_exit_procedure: amt$label_exit_procedure,  
= amc$label_options =  
  label_options: amt$label_options,  
= amc$label_type =  
  label_type: amt$label_type,  
= amc$line_number =  
  line_number: amt$line_number,
```

```

= amc$max_block_length =
  max_block_length: amt$max_block_length,
= amc$max_record_length =
  max_record_length: amt$max_record_length,
= amc$min_block_length =
  min_block_length: amt$min_block_length,
= amc$min_record_length =
  min_record_length: amt$min_record_length,
= amc$null_attribute =
  ,
= amc$open_position =
  open_position: amt$open_position,
= amc$padding_character =
  padding_character: amt$padding_character,
= amc$page_format =
  page_format: amt$page_format,
= amc$page_length =
  page_length: amt$page_length,
= amc$page_width =
  page_width: amt$page_width,
= amc$permanent_file =
  permanent_file: boolean,
= amc$preset_value =
  preset_value: amt$preset_value,
= amc$record_type =
  record_type: amt$record_type,
= amc$ring_attributes =
  ring_attributes: amt$ring_attributes,
= amc$statement_identifier =
  statement_identifier: amt$statement_identifier,
= amc$user_info =
  user_info: amt$user_info,
= amc$average_record_length =
  average_record_length:
    amt$average_record_length,
= amc$collate_table =
  collate_table: ^amt$collate_table,
= amc$collate_table_name =
  collate_table_name: pmt$program_name,
= amc$compression_procedure_name =
  compression_procedure_name: [input,output]
    ^amt$compression_procedure_name,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$data_padding =
  data_padding: amt$data_padding,
= amc$dynamic_home_block_space =
  dynamic_home_block_space:
    amt$dynamic_home_block_space,
= amc$embedded_key =
  embedded_key: boolean,
= amc$error_limit =
  error_limit: amt$error_limit,
= amc$estimated_record_count =
  estimated_record_count:
    amt$estimated_record_count,
= amc$hashing_procedure_name =
  hashing_procedure_name: [input,output]
    ^amt$hashing_procedure_name,
= amc$index_levels =
  index_levels: amt$index_levels,
= amc$index_padding =
  index_padding: amt$index_padding,
= amc$initial_home_block_count =
  initial_home_block_count:
    amt$initial_home_block_count,
= amc$key_length =
  key_length: amt$key_length,
= amc$key_position =
  key_position: amt$key_position,
= amc$key_type =
  key_type: amt$key_type,
= amc$loading_factor =
  loading_factor: amt$loading_factor,
= amc$lock_expiration_time =
  lock_expiration_time: amt$lock_expiration_time,
= amc$logging_options =
  logging_options: amt$logging_options,
= amc$log_residence =
  log_residence: [input,output]
    ^amt$log_residence,
= amc$message_control =
  message_control: amt$message_control,
= amc$record_limit =
  record_limit: amt$record_limit,
= amc$records_per_block =
  records_per_block: amt$records_per_block,
casend,
recend;
```

```

amt$file_access_code = char { defaults to space },
amt$file_access_selections = ^array [1 .. * ] of
  amt$access_selection,
amt$file_attribute_keys = 1 .. amc$max_attribute,
amt$file_attributes = array [1 .. * ] of
  amt$file_item,
amt$file_byte_address = 0 .. amc$file_byte_limit;
amt$file_contents = ost$name;
amt$file_id_ordinal = 0 .. amc$max_file_id_ordinal,
amt$file_id_sequence = 1 .. 4095;
amt$file_identifier = record
  ordinal: amt$file_id_ordinal,
  sequence: amt$file_id_sequence,
  recend,
amt$file_item = record
  case key { input } : amt$file_attribute_keys of
  { input }
    = amc$access_mode =
      access_mode: pft$usage_selections,
    = amc$block_type =
      block_type: amt$block_type,
    = amc$character_conversion =
      character_conversion: boolean,
    = amc$clear_space =
      clear_space: ost$clear_file_space,
    = amc$error_exit_name =
      error_exit_name: pmt$program_name,
    = amc$error_options =
      error_options: amt$error_options,
    = amc$file_access_procedure =
      file_access_procedure: pmt$program_name,
    = amc$file_contents =
      file_contents: amt$file_contents,
    = amc$file_limit =
      file_limit: amt$file_limit,
    = amc$file_organization =
      file_organization: amt$file_organization,
    = amc$file_processor =
      file_processor: amt$file_processor,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$file_structure =  
  file_structure: amt$file_structure,  
= amc$forced_write =  
  forced_write: amt$forced_write,  
= amc$internal_code =  
  internal_code: amt$internal_code,  
= amc$label_exit_name =  
  label_exit_name: pmt$program_name,  
= amc$label_options =  
  label_options: amt$label_options,  
= amc$label_type =  
  label_type: amt$label_type,  
= amc$line_number =  
  line_number: amt$line_number,  
= amc$max_block_length =  
  max_block_length: amt$max_block_length,  
= amc$max_record_length =  
  max_record_length: amt$max_record_length,  
= amc$min_block_length =  
  min_block_length: amt$min_block_length,  
= amc$min_record_length =  
  min_record_length: amt$min_record_length,  
= amc$null_attribute =  
  ,  
= amc$open_position =  
  open_position: amt$open_position,  
= amc$padding_character =  
  padding_character: amt$padding_character,  
= amc$page_format =  
  page_format: amt$page_format,  
= amc$page_length =  
  page_length: amt$page_length,  
  
= amc$page_width =  
  page_width: amt$page_width,  
= amc$preset_value =  
  preset_value: amt$preset_value,  
= amc$record_type =  
  record_type: amt$record_type,  
= amc$return_option =  
  return_option: amt$return_option,  
= amc$ring_attributes =  
  ring_attributes: amt$ring_attributes,  
= amc$statement_identifier =  
  statement_identifier: amt$statement_identifier,
```

```

= amc$user_info =
  user_info: amt$user_info,
= amc$vertical_print_density =
  vertical_print_density:
    amt$vertical_print_density,
= amc$average_record_length =
  average_record_length:
    amt$average_record_length,
= amc$collate_table_name =
  collate_table_name: pmt$program_name,
= amc$compression_procedure_name =
  compression_procedure_name: [input,output]
    ^amt$compression_procedure_name,
= amc$data_padding =
  data_padding: amt$data_padding,
= amc$dynamic_home_block_space =
  dynamic_home_block_space: amt$dynamic_home_block_space,
= amc$embedded_key =
  embedded_key: boolean,
= amc$error_limit =
  error_limit: amt$error_limit,
= amc$estimated_record_count =
  estimated_record_count:
    amt$estimated_record_count,
= amc$hashing_procedure_name =
  hashing_procedure_name: [input,output]
    ^amt$hashing_procedure_name,
= amc$index_levels =
  index_levels: amt$index_levels,
= amc$index_padding =
  index_padding: amt$index_padding,
= amc$initial_home_block_count =
  initial_home_block_count: amt$initial_home_block_count,
= amc$key_length =
  key_length: amt$key_length,
= amc$key_position =
  key_position: amt$key_position,
= amc$key_type =
  key_type: amt$key_type,
= amc$loading_factor =
  loading_factor: amt$loading_factor,
= amc$lock_expiration_time =
  lock_expiration_time: amt$lock_expiration_time,
= amc$logging_options =
  logging_options: amt$logging_options,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$log_residence =
  log_residence: [input,output] ^amt$log_residence,
= amc$message_control =
  message_control: amt$message_control,
= amc$record_limit =
  record_limit: amt$record_limit,
= amc$records_per_block =
  records_per_block: amt$records_per_block,
casend,
recend;

amt$file_label_field = record
case key { input } : amt$file_label_keys of { input }
= amc$file_label_id =
  { identifies file within volume }
  file_label_id: amt$file_label_id,
= amc$file_set_id =
{ identifies this file_set among others }
  file_set_id: amt$file_set_id,
= amc$section_number =
{ identifies this section among the sections of }
{ this file. }
  section_number: amt$section_number,
= amc$sequence_number =
{ identifies this file among the files of this }
{ file set. }
  sequence_number: amt$sequence_number,
= amc$generation_number =
  generation_number: amt$generation_number,
= amc$version_number =
  version_number: amt$version_number,
= amc$creation_date =
  creation_date: amt$creation_date,
= amc$expiration_date =
  expiration_date: amt$expiration_date,
= amc$file_access_code =
{ Indicates restrictions on access to the file. }
{ Space means no access restrictions }
  file_access_code: amt$file_access_code,
casend,
recend;
```



```

amt$file_label_fields = array [ * ] of
    amt$file_label_field;

amt$file_label_id = string (17), { defaults to }
{ spaces };

amt$file_label_keys = (amc$file_label_id,
    amc$file_set_id, amc$section_number,
    amc$sequence_number, amc$generation_number,
    amc$version_number, amc$creation_date,
    amc$expiration_date, amc$file_access_code);

amt$file_lock = (amc$lock_set, amc$already_set);

amt$file_length = 0 .. amc$file_byte_limit;

amt$file_limit = 0 .. amc$file_byte_limit;

amt$file_organization = (amc$sequential, amc$byte_addressable,
    amc$indexed_sequential, amc$direct_access, amc$system_key);

amt$file_position = (amc$boi, amc$bop,
    amc$mid_record, amc$eor, amc$eop, amc$eoi);

amt$file_processor = ost$name;

amt$file_reference = string ( * <= amc$max_path_name_size);

amt$file_set_id = string (6), { defaults to spaces };

amt$file_structure = ost$name;

amt$find_record_space = record
    space: amt$file_length,
    where: amt$put_locality,
    wait: ost$wait,
recend;

amt$forced_write = (amc$forced,
    amc$forced_if_structure_change, amc$unforced),

amt$general_commit = record
    case general_commit_in_use: boolean of
    = TRUE =
        general_commit_name: ost$name,
    casend,
recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$generation_number = 1 .. 9999 { defaults }
{ to 0001 };
amt$get_attributes = array [1 .. * ] of
  amt$get_item;

amt$get_item = record
  source { output } : amc$undefined_attribute ..
    amc$access_method_default,
  case key { input } : amt$file_attribute_keys of
{ output }
  = amc$access_mode =
    access_mode: pft$usage_selections,
  = amc$application_info =
    application_info: pft$application_info,
  = amc$block_type =
    block_type: amt$block_type,

  = amc$character_conversion =
    character_conversion: boolean,
  = amc$clear_space =
    clear_space: ost$clear_file_space,
  = amc$error_exit_name =
    error_exit_name: pmt$program_name,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$file_access_procedure =
    file_access_procedure: pmt$program_name,
  = amc$file_contents =
    file_contents: amt$file_contents,
  = amc$file_length =
    file_length: amt$file_length,
  = amc$file_limit =
    file_limit: amt$file_limit,
  = amc$file_organization =
    file_organization: amt$file_organization,
  = amc$file_processor =
    file_processor: amt$file_processor,
  = amc$file_structure =
    file_structure: amt$file_structure,
  = amc$forced_write =
    forced_write: amt$forced_write,
  = amc$global_access_mode =
    global_access_mode: pft$usage_selections,
  = amc$global_file_address =
    global_file_address: amt$file_byte_address,
  = amc$global_file_name =
    global_file_name: ost$binary_unique_name,
```

```

= amc$global_file_position =
  global_file_position: amt$global_file_position,
= amc$global_share_mode =
  global_share_mode: pft$share_selections,
= amc$internal_code =
  internal_code: amt$internal_code,
= amc$label_exit_name =
  label_exit_name: pmt$program_name,
= amc$label_options =
  label_options: amt$label_options,
= amc$label_type =
  label_type: amt$label_type,
= amc$line_number =
  line_number: amt$line_number,
= amc$max_block_length =
  max_block_length: amt$max_block_length,
= amc$max_record_length =
  max_record_length: amt$max_record_length,
= amc$min_block_length =
  min_block_length: amt$min_block_length,
= amc$min_record_length =
  min_record_length: amt$min_record_length,
= amc$null_attribute =
  ,
= amc$open_position =
  open_position: amt$open_position,
= amc$padding_character =
  padding_character: amt$padding_character,

= amc$page_format =
  page_format: amt$page_format,
= amc$page_length =
  page_length: amt$page_length,
= amc$page_width =
  page_width: amt$page_width,
= amc$permanent_file =
  permanent_file: boolean,
= amc$preset_value =
  preset_value: amt$preset_value,
= amc$record_type =
  record_type: amt$record_type,
= amc$return_option =
  return_option: amt$return_option,
= amc$ring_attributes =
  ring_attributes: amt$ring_attributes,
= amc$statement_identifier =
  statement_identifier: amt$statement_identifier,
= amc$user_info =
  user_info: amt$user_info,

```

CONSTANT AND TYPE DECLARATIONS

```
= amc$vertical_print_density =  
  vertical_print_density:  
    amt$vertical_print_density,  
= amc$average_record_length =  
  average_record_length:  
    amt$average_record_length,  
= amc$collate_table_name =  
  collate_table_name: pmt$program_name,  
= amc$data_padding =  
  data_padding: amt$data_padding,  
= amc$embedded_key =  
  embedded_key: boolean,  
= amc$error_limit =  
  error_limit: amt$error_limit,  
= amc$estimated_record_count =  
  estimated_record_count:  
    amt$estimated_record_count,  
= amc$index_levels =  
  index_levels: amt$index_levels,  
= amc$index_padding =  
  index_padding: amt$index_padding,  
= amc$key_length =  
  key_length: amt$key_length,  
= amc$key_position =  
  key_position: amt$key_position,  
= amc$key_type =  
  key_type: amt$key_type,  
= amc$message_control =  
  message_control: amt$message_control,  
= amc$record_limit =  
  record_limit: amt$record_limit,  
= amc$records_per_block =  
  records_per_block: amt$records_per_block,  
  casend  
recend;  
  
amt$get_key_definitions = record  
  key_definitions: ^SEQ (*),  
recend;
```

```

amt$get_lock_keyed_record = record
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  key_location: ^cell,
  major_key_length: amt$major_key_length,
  relation: amt$key_relation,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

```

```

amt$get_lock_next_keyed_record = record
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  key_location: ^cell,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

```

```

amt$get_nested_file_definitions = record
  definitions: ^amt$nested_file_definitions,
  nested_file_count: ^amt$nested_file_count,
recend;

```

```

amt$get_next_primary_key_list = record
  high_key: ^cell,
  major_high_key: amt$major_key_length,
  high_key_relation: amt$key_relation,
  working_storage_area: ^cell,
  working_storage_length: amt$working_storage_length,
  end_of_primary_key_list: ^boolean,
  transferred_byte_count: ^amt$working_storage_length,
  transferred_key_count: ^amt$key_count_limit,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$get_primary_key_count = record
  low_key: ^cell,
  major_low_key: amt$major_key_length,
  low_key_relation: amt$key_relation,
  high_key: ^cell,
  major_high_key: amt$major_key_length,
  high_key_relation: amt$key_relation,
  list_count_limit: amt$key_count_limit,
  list_count: ^amt$key_count_limit,
  wait: ost$wait,
recend;

amt$global_file_position = amt$file_position;

amt$group_name = amt$key_name;

amt$hashing_procedure = ^procedure (old_key: ^cell;
  key_length: amt$key_length;
  VAR hashed_key: integer;
  VAR status: ost$status);

amt$hashing_procedure_name = amt$entry_point_reference;

amt$index_levels = 0 .. amc$max_index_level;

amt$index_padding = 0 .. 99 { expressed as a }
{ percentage };

amt$initial_home_block_count = 1 .. amc$max_home_blocks;

amt$internal_code = (amc$as6, amc$as8, amc$ascii,
  amc$d64, amc$ebcdic, amc$bcd);

amt$key_count_limit = 0 .. amc$file_byte_limit;

amt$key_length = 1 .. amc$max_key_length;

amt$key_name = ost$name;

amt$key_position = 0 .. amc$max_key_position;

amt$key_relation = (amc$equal_key,
  amc$greater_or_equal_key, amc$greater_key);

amt$key_type = (amc$collated_key, amc$integer_key,
  amc$uncollated_key);
```

```

amt$label_area_length = 18 .. amc$max_label_length;

amt$label_exit_procedure = ^procedure
  (file_identifier: amt$file_identifier);

amt$label_options = set of (amc$vol1, amc$uvl,
  amc$hdr1, amc$hdr2, amc$eov1, amc$eov2, amc$uhl,
  amc$eof1, amc$eof2, amc$utl);

amt$label_type = (amc$labelled,
  amc$non_standard_labelled, amc$unlabelled);

amt$last_access_operation = amc$last_access_start ..
  amc$max_operation;

amt$last_op_status = (amc$active, amc$complete);

amt$last_operation = 1 .. amc$max_operation;

amt$line_number = record
  length: amt$line_number_length,
  location: amt$line_number_location,
  recend;

amt$line_number_length = 1 .. amc$max_line_number;

amt$line_number_location = amt$page_width;

amt$loading_factor = 0 .. 100;

amt$local_file_name = ost$name;

amt$lock_expiration_time = 0 .. 604800000 { milliseconds } ;

amt$lock_intent = (amc$exclusive_access, amc$preserve_access_
  and_content, amc$preserve_content);

amt$lock_file = record
  wait_for_lock: ost$wait_for_lock,
  lock_intent: amt$lock_intent,
  recend;

amt$lock_key = record
  key_location: ^cell,
  wait_for_lock: ost$wait_for_lock,
  unlock_control: amt$unlock_control,
  lock_intent: amt$lock_intent,
  recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$logging_options = set of amt$logging_possibilities;

amt$logging_possibilities = (amc$enable_parcel, amc$enable_
    media_recovery, amc$enable_request_recovery);

amt$log_residence = amt$path_name;

amt$major_key_length = 0 .. amc$max_key_length;

amt$max_block_length = 1 .. amc$maximum_block - 1;

amt$max_optional_attributes = 1 .. amc$max_optional_attributes;

amt$max_record_length = 0 .. amc$maximum_record;

amt$max_repeating_group_count = amt$max_record_length;

amt$message_control = set of (amc$trivial_errors,
    amc$messages, amc$statistics);

amt$min_block_length = 18 .. amc$maximum_block;

amt$min_record_length = 0 .. amc$maximum_record;

amt$nested_file_definition = record
    nested_file_name: amt$nested_file_name,
    embedded_key: boolean,
    key_position: amt$key_position,
    key_length: amt$key_length,
    maximum_record: amt$max_record_length,
    minimum_record: amt$min_record_length,
    record_type: amt$record_type,
    case file_organization: amt$file_organization of
    = amc$indexed_sequential =
        key_type: amt$key_type,
        collate_table_name: pmt$program_name,
        data_padding: amt$data_padding,
        index_padding: amt$index_padding,
    = amc$direct_access =
        home_block_count: amt$initial_home_block_count,
        dynamic_home_block_space: amt$dynamic_home_block_space,
        loading_factor: amt$loading_factor,
        hashing_procedure: amt$hashing_procedure_name,
    = amc$system_key =
        records_per_block: amt$records_per_block,
    casend,
recend;
```



```

amt$nested_file_definitions = array [1 .. * ] of
    amt$nested_file_definition;

amt$nested_file_count = 1 .. amc$max_blocks_per_file;

amt$nested_file_name = ost$name;

amt$nowait_var_parameters = SEQ (REP 10 of integer);

amt$open_position = (amc$open_no_positioning,
    amc$open_at_boi, amc$open_at_bop, amc$open_at_eoi);

amt$optional_key_attribute = record
    case selector: amt$file_attribute_keys of
    = amc$key_type =
        key_type: amt$key_type,
    = amc$collate_table_name =
        collate_table_name: pmt$program_name,
    = amc$duplicate_keys =
        duplicate_key_control: amt$duplicate_key_control,
    = amc$null_suppression =
        null_suppression: boolean,
    = amc$sparse_keys =
        sparse_key_control_position: amt$key_position,
        sparse_key_control_characters: set of char,
        sparse_key_control_effect: amt$sparse_key_control_effect,
    = amc$repeating_group =
        repeating_group_length: amt$max_record_length,
        repetition_control: amt$repetition_control,
    = amc$concatenated_key_portion =
        concatenated_key_position: amt$key_position,
        concatenated_key_length: amt$key_length,
        concatenated_key_type: amt$key_type,
    = amc$group_name =
        group_name: amt$group_name,
    = amc$variable_length_key =
        key_delimiter_characters: set of char,
        casend,
    recend;

amt$optional_key_attributes = array [1 .. * ] of
    amt$optional_key_attribute;

amt$pack_block_header = record
    header_type: amt$block_header_type,
    block_length: amt$max_block_length,
    block_number: amt$block_number,
    unused_bit_count: amt$unused_bit_count,
    recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$padding_character = char;

amt$page_format = (amc$continuous_form,
  amc$burstable_form, amc$non_burstable_form);

amt$page_length = 1 .. amc$file_byte_limit,
amt$page_width = 1 .. amc$max_page_width;

amt$path_name = string (amc$max_path_name_size);

amt$physical_transfer_count = 0 ..
  amc$max_buffer_length;

amt$pointer_kind = (amc$cell_pointer,
  amc$heap_pointer, amc$sequence_pointer);

amt$preset_value = integer;

amt$primary_key = ^cell;

amt$put_locality = (amc$put_near_anywhere, amc$put_near_get,
  amc$put_near_update);

amt$record_header = record
  header_type: amt$record_header_type,
  length: amt$max_record_length,
  previous_length: amt$max_record_length,
  unused_bit_count: amt$unused_bit_count,
  user_information: cell,
recend;

amt$record_header_length = 0 ..
  amc$max_record_header;

amt$record_header_type = (amc$full_record,
  amc$start_record, amc$continued_record,
  amc$end_record, amc$partition,
  amc$deleted_record);

amt$record_limit = 1 .. amc$file_byte_limit;

amt$record_type = (amc$variable { V } ,
  amc$undefined { U } , amc$sansi_fixed { F } ,
  amc$sansi_spanned { S } , amc$sansi_variable { D } );

amt$records_per_block = 1 .. amc$max_records_per_block;
```

```

amt$recovered_request = record
  past_last: boolean,
  task_id: pmt$task_id,
  file_identifier: amt$file_identifier,
  nested_file_selection: amt$nested_file_name,
  call_block: amt$call_block,
  status: ost$status,
  working_storage_length: amt$working_storage_length,
  key_length: amt$key_length,
recend;

amt$recovery_description = record
  case recover_option: amt$recovery_options of
  = amc$recover_file_media =
    media_recovery: record
      backup_date_time: ost$date_time,
      last_requests: ^SEQ ( * ),
    recend,
  = amc$recover_to_last_requests =
    last_requests: ^SEQ ( * ),
  = amc$recover_file_structure =
    ,
  = amc$salvage_data_records =
    new_keyed_file: amt$local_file_name,
    salvage_log: amt$salvage_log_description,
  casend,
recend;

amt$recovery_options = (amc$recover_file_media,
  amc$recover_to_last_requests, amc$recover_file_structure,
  amc$salvage_data_records);

amt$repetition_control = record
  case repeat_to_end_of_record: boolean of
  = FALSE =
    repeating_group_count: amt$max_repeating_group_count,
  casend,
recend;

amt$residual_skip_count = amt$skip_count;

amt$return_option = (amc$return_at_close,
  amc$return_at_task_exit, amc$return_at_job_exit);

```

CONSTANT AND TYPE DECLARATIONS

```
amt$ring_attributes = record
  r1: ost$valid_ring,
  r2: ost$valid_ring,
  r3: ost$valid_ring,
recend;

amt$salvage_log_description = record
  case salvage_log_wanted: boolean of
    = TRUE =
      rejects_file: amt$local_file_name,
      casend,
recend;

amt$section_number = 1 .. 9999,
{ defaults to 0001 };

amt$segment_pointer = record
  case kind: amt$pointer_kind of
    =amc$cell_pointer=
      cell_pointer: ^cell,
    =amc$heap_pointer=
      heap_pointer: ^HEAP (*),
    =amc$sequence_pointer=
      sequence_pointer: ^SEQ (*),
      casend,
recend;

amt$select_key = record
  key_name: amt$key_name,
recend;

amt$select_nested_file = record
  nested_file_name: amt$nested_file_name,
recend;

amt$separate_key_groups = record
  group: amt$group_name,
  parallel_group: amt$group_name,
recend;
```

```

amt$selected_key_name = amt$key_name;
amt$selected_nested_file = amt$nested_file_name;
amt$sequence_number = 1 .. 9999
{ defaults to 0001 };
amt$skip_buffer_length = 1 .. amc$max_buffer_length;
amt$skip_count = 0 .. amc$file_byte_limit;
amt$skip_direction = (amc$forward, amc$backward);
amt$skip_option = (amc$skip_to_eor, amc$no_skip);
amt$skip_unit = (amc$skip_record, amc$skip_block,
  amc$skip_partition);
amt$sparse_key_control_effect = (amc$include_key_value,
  amc$exclude_key_value);
amt$statement_id_length = 1 ..
  amc$max_statement_id_length;
amt$statement_id_location = amt$page_width;
amt$statement_identifier = record
  length: amt$statement_id_length,
  location: amt$statement_id_location,
  recend;

amt$store_attributes = array [1 .. * ] of
  amt$store_item;

amt$store_item = record
  case key: amt$file_attribute_keys of
  = amc$error_exit_procedure =
    error_exit_procedure: amt$error_exit_procedure,
  = amc$error_options =
    error_options: amt$error_options,
  = amc$label_exit_procedure =
    label_exit_procedure: amt$label_exit_procedure,
  = amc$label_options =
    label_options: amt$label_options,
  = amc$null_attribute =
    ,
  = amc$error_limit =
    error_limit: amt$error_limit,
  = amc$message_control =
    message_control: amt$message_control,
  casend,
  recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$tape_mark_count = 1 .. amc$max_tape_mark_count;

amt$term_option = (amc$start, amc$continue,
  amc$terminate);

amt$transfer_count = amt$working_storage_length;

amt$unpack_block_header = record
  header_type: amt$block_header_type,
  block_length_as_read: amt$max_block_length,
  block_length_as_written: amt$max_block_length,
  block_number: amt$block_number,
  unused_bit_count: amt$unused_bit_count,
  block_status: amt$block_status,
recend;

amt$unused_bit_count = 0 .. 7;

amt$user_info = string (amc$max_user_info);

amt$version_number = 1 .. 99
{ defaults to 01 };

amt$vertical_print_density = 6 ..
  amc$max_lines_per_inch;

amt$volume_number = 1 .. amc$max_vol_number;

amt$volume_position = (amc$bov,
  amc$mid_bov_label_group, amc$after_tapemark,
  amc$mid_hdr_label_group, amc$mid_eof_label_group,
  amc$mid_eov_label_group, amc$eov);

amt$working_storage_length = ost$segment_length;
```

FAP Call Block Declarations

```
amt$check_buffer_req = record
  buffer_area: ^cell,
  request_complete: ^boolean,
  byte_address: ^amt$file_byte_address,
  transfer_count: ^amt$physical_transfer_count,
  wait: ost$wait,
recend;
```

```
amt$check_record_req = record
  working_storage_area: ^cell,
  request_complete: ^boolean,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;
```

```
amt$delete_direct_req = record
  byte_address: amt$file_byte_address,
recend;
```

```
amt$delete_key_req = record
  key_location: ^cell,
  wait: ost$wait,
recend;
```

```
amt$fetch_access_information_rq = record
  access_information: ^amt$access_information,
recend;
```

```
amt$fetch_req = record
  file_attributes: ^amt$fetch_attributes,
recend;
```

```
amt$flush_req = record
  wait: ost$wait,
recend;
```

```
amt$get_direct_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  transfer_count: ^amt$transfer_count,
  byte_address: amt$file_byte_address,
  file_position: ^amt$file_position,
recend;
```

CONSTANT AND TYPE DECLARATIONS

```
amt$get_key_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  key_location: ^cell,
  major_key_length: amt$major_key_length,
  key_relation: amt$key_relation,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

amt$get_label_req = record
  label_area: ^cell,
  label_area_length: amt$label_area_length,
recend;

amt$get_next_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  transfer_count: ^amt$transfer_count,
  byte_address: ^amt$file_byte_address,
  file_position: ^amt$file_position,
recend;

amt$get_next_key_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  key_location: ^cell,
  record_length: ^amt$max_record_length,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

amt$get_space_used_for_key = record
  low_key: ^cell,
  major_low_key: amt$major_key_length,
  low_key_relation: amt$key_relation,
  high_key: ^cell,
  major_high_key: amt$major_key_length,
  high_key_relation: amt$key_relation,
  data_block_count: ^amt$data_block_count,
  data_block_space: ^amt$file_length,
  wait: ost$wait,
recend;
```



```

amt$get_partial_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  record_length: ^amt$max_record_length,
  transfer_count: ^amt$transfer_count,
  byte_address: ^amt$file_byte_address,
  file_position: ^amt$file_position,
  skip_option: amt$skip_option,
recend;

```

```

amt$get_segment_pointer_req = record
  pointer_kind: amt$pointer_kind,
  segment_pointer: ^amt$segment_pointer,
recend;

```

```

amt$lock_file_req = record
  status: ^amt$file_lock,
recend;

```

```

amt$open_req = record
  local_file_name: amt$local_file_name,
  access_level: amt$access_level,
  existing_file: boolean,
  contains_data: boolean,
recend;

```

```

amt$pack_block_req = record
  buffer_area: amt$buffer_area,
  header: amt$pack_block_header,
recend;

```

```

amt$pack_record_req = record
  buffer_area: amt$buffer_area,
  header: amt$record_header,
recend;

```

```

amt$put_direct_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  byte_address: amt$file_byte_address,
recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$put_key_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  key_location: ^cell,
  wait: ost$wait,
recend;
```

```
amt$put_label_req = record
  label_area: ^cell,
  label_area_length: amt$label_area_length,
recend;
```

```
amt$put_next_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  byte_address: ^amt$file_byte_address,
recend;
```

```
amt$put_partial_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  byte_address: ^amt$file_byte_address,
  term_option: amt$term_option,
recend;
```

```
amt$putrep_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  key_location: ^cell,
  wait: ost$wait,
recend;
```

```
amt$read_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: ^amt$file_byte_address,
  transfer_count: ^amt$physical_transfer_count,
  wait: ost$wait,
recend;
```

```

amt$read_direct_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: amt$file_byte_address,
  transfer_count: ^amt$physical_transfer_count,
  wait: ost$wait,
recend;

```

```

amt$read_direct_skip_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: amt$file_byte_address,
  transfer_count: ^amt$physical_transfer_count,
  wait: ost$wait,
recend;

```

```

amt$read_skip_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: ^amt$file_byte_address,
  transfer_count: ^amt$physical_transfer_count,
  wait: ost$wait,
recend;

```

```

amt$replace_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
recend;

```

```

amt$replace_direct_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  byte_address: amt$file_byte_address,
recend;

```

```

amt$replace_key_req = record
  working_storage_area: ^cell,
  working_storage_length:
    amt$working_storage_length,
  key_location: ^cell,
  wait: ost$wait,
recend;

```

```

amt$rewind_req = record
  wait: ost$wait,
recend;

```

CONSTANT AND TYPE DECLARATIONS

```
amt$rewind_volume_req = record
  wait: ost$wait,
recend;

amt$salvage_log_description = record
  case salvage_log_wanted: boolean of
    = TRUE =
      rejects_file: amt$local_file_name,
      casend,
recend;

amt$seek_direct_req = record
  byte_address: amt$file_byte_address,
recend;

amt$set_segment_eoi_req = record
  segment_pointer: amt$segment_pointer,
recend;

amt$set_segment_position_req = record
  segment_pointer: amt$segment_pointer,
recend;

amt$skip_req = record
  direction: amt$skip_direction,
  unit: amt$skip_unit,
  count: amt$skip_count,
  file_position: ^amt$file_position,
recend;

amt$start_req = record
  key_location: ^cell,
  major_key_length: amt$major_key_length,
  key_relation: amt$key_relation,
  file_position: ^amt$file_position,
  wait: ost$wait,
recend;

amt$store_req = record
  file_attributes: ^amt$store_attributes,
recend;

amt$unlock_key = record
  unlock_all_keys: boolean,
  key_location: ^cell,
recend;
```

```
amt$unpack_block_req = record
  buffer_area: amt$buffer_area,
  header: ^amt$unpack_block_header,
recend;
```

```
amt$unpack_record_req = record
  buffer_area: amt$buffer_area,
  header: ^amt$record_header,
recend;
```

```
amt$write_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: ^amt$file_byte_address,
  wait: ost$wait,
recend;
```

```
amt$write_direct_req = record
  buffer_area: ^cell,
  buffer_length: amt$buffer_length,
  byte_address: amt$file_byte_address,
  wait: ost$wait,
recend;
```

```
ift$fetch_terminal_req = record
  terminal_attributes: ^ift$get_terminal_attributes,
recend;
```

```
ift$store_terminal_req = record
  terminal_attributes:
    ^ift$store_terminal_attributes,
recend;
```

AV

```
avt$account_name = ost$name;
```

```
avt$project_name = ost$name;
```

IF

Constants

```
ifc$interactive_facility_id = 'IF';
ifc$max_collector_string_size = 31 { * } ;
ifc$max_ecc = ifc$min_ecc + 500;
ifc$max_file_mark_string_size = 31;
ifc$max_prompt_string_size = 31;
ifc$max_transparent_count_size = 4096;
ifc$min_ecc = 320000;
```

Types

```
ift$attribute_source = (ifc$undefined_attribute,
  ifc$nam_default, ifc$os_default,
  ifc$terminal_command, ifc$terminal_request,
  ifc$request_terminal_command,
  ifc$request_terminal_request,
  ifc$store_terminal_request, ifc$bam_request);
```

```
ift$carriage_return_idle_range = 0 .. 99;
```

```
ift$collector_delimiter = record
  relation: ift$collector_relations,
  delimiter_string: ift$collector_delimiter_string,
recend;
```

```
ift$collector_delimiter_string = record
  size: ift$collector_string_size,
  value: string (ifc$max_collector_string_size),
recend;
```

```

ift$collector_relations = (ifc$less_relation,
    ifc$less_equal_relation, ifc$equal_relation,
    ifc$not_equal_relation, ifc$greater_relation,
    ifc$greater_equal_relation);

ift$collector_string_size = 1 ..
    ifc$max_collector_string_size;

ift$file_mark_string = record
    size: ift$file_mark_string_size,
    value: string (ifc$max_file_mark_string_size),
recend;

ift$file_mark_string_size = 0 ..
    ifc$max_file_mark_string_size;

ift$get_terminal_attributes = array [1 .. * ] of
    ift$get_terminal_attribute;

ift$get_terminal_attribute = record
    source { output } : ift$attribute_source,
    case key { input } : ift$terminal_attribute_keys of
    = ifc$abort_line_character =
        abort_line_character: char,
    = ifc$auto_input_mode =
        auto_input_mode: boolean,
    = ifc$backspace_character =
        backspace_character: char,
    = ifc$buffer_behind { * } =
        buffer_behind: boolean,
    = ifc$cancel_group_character =
        cancel_group_character: char,
    = ifc$cancel_line_character =
        cancel_line_character: char,
    = ifc$carriage_return_idle =
        carriage_return_idle:
            ift$carriage_return_idle_range,
    = ifc$collector_delimiter { * } =
        collector_delimiter: ift$collector_delimiter,
    = ifc$collector_mode { * } =
        collector_mode: boolean,
    = ifc$echoplex =
        echoplex: boolean,
    = ifc$eoi_string =
        eoi_string: ift$file_mark_string,
    = ifc$eop_string =
        eop_string: ift$file_mark_string,

```

CONSTANT AND TYPE DECLARATIONS

```
= ifc$input_device =
  input_device: ift$input_devices,
= ifc$line_feed_idle =
  line_feed_idle: ift$line_feed_idle_range,
= ifc$network_control_character =
  network_control_character: char,
= ifc$no_format_effectors =
  no_format_effectors: boolean,
= ifc$null_attribute =
  ,
= ifc$output_device =
  output_device: ift$output_devices,
= ifc$output_flow_control =
  output_flow_control: boolean,
= ifc$output_translation =
  output_translation: pmt$program_name,
= ifc$page_length =
  page_length: amt$page_length,
= ifc$page_wait =
  page_wait: boolean,
= ifc$page_width =
  page_width: amt$page_width,
= ifc$parity =
  parity: ift$parity_modes,
= ifc$pause_break_character =
  pause_break_character: char,
= ifc$prompt_file =
  prompt_file: amt$local_file_name,
= ifc$prompt_file_id =
  prompt_file_id: amt$file_identifier,
= ifc$prompt_string =
  prompt_string: ift$prompt_string,
= ifc$special_editing =
  special_editing: boolean,
= ifc$terminal_class =
  terminal_class: ift$terminal_classes,
= ifc$terminal_name =
  terminal_name: string (7),
= ifc$terminate_break_character =
  terminate_break_character: char,
= ifc$transparent_delim_selection =
  transparent_delim_selection:
    ift$transparent_delim_selection,
= ifc$transparent_end_character =
  transparent_end_character: char,
```



```

= ifc$transparent_end_count =
  transparent_end_count:
    ift$transparent_count_range,
= ifc$transparent_mode =
  transparent_mode: boolean,
= ifc$type_ahead =
  type_ahead: boolean,
  casend,
recend;

ift$input_devices = (ifc$keyboard_input,
  ifc$paper_tape_input);

ift$line_feed_idle_range = 0 .. 99;

ift$output_devices = (ifc$display_output,
  ifc$printer_output, ifc$paper_tape_output);

ift$parity_modes = (ifc$no_parity, ifc$even_parity,
  ifc$odd_parity, ifc$zero_parity);

ift$prompt_string = record
  size: ift$prompt_string_size,
  value: string (ifc$max_prompt_string_size),
recend;

ift$prompt_string_size = 0 ..
  ifc$max_prompt_string_size;

ift$req_terminal_req_attribute = record
  case key { input } : ift$terminal_attribute_keys of
= ifc$auto_input_mode =
  auto_input_mode: boolean,
= ifc$carriage_return_idle =
  carriage_return_idle:
    ift$carriage_return_idle_range,
= ifc$collector_delimiter { * } =
  collector_delimiter: ift$collector_delimiter,
= ifc$collector_mode { * } =
  collector_mode: boolean,
= ifc$echoplex =
  echoplex: boolean,

```

CONSTANT AND TYPE DECLARATIONS

```
= ifc$eoi_string =
  eoi_string: ift$file_mark_string,
= ifc$eop_string =
  eop_string: ift$file_mark_string,
= ifc$input_device =
  input_device: ift$input_devices,
= ifc$line_feed_idle =
  line_feed_idle: ift$line_feed_idle_range,
= ifc$no_format_effectors =
  no_format_effectors: boolean,
= ifc$null_attribute =
  ,
= ifc$output_device =
  output_device: ift$output_devices,
= ifc$output_translation =
  output_translation: pmt$program_name,
= ifc$page_wait =
  page_wait: boolean,
= ifc$prompt_file =
  prompt_file: amt$local_file_name,
= ifc$prompt_file_id =
  prompt_file_id: amt$file_identifier,
= ifc$prompt_string =
  prompt_string: ift$prompt_string,
= ifc$special_editing =
  special_editing: boolean,
= ifc$transparent_delim_selection =
  transparent_delim_selection:
    ift$transparent_delim_selection,
= ifc$transparent_end_character =
  transparent_end_character: char,
= ifc$transparent_end_count =
  transparent_end_count:
    ift$transparent_count_range,
= ifc$transparent_mode =
  transparent_mode: boolean,
= ifc$type_ahead =
  type_ahead: boolean,
casend,
recend;

ift$req_terminal_req_attributes = array [1 .. * ] of
  ift$req_terminal_req_attribute;

ift$store_terminal_attributes = array [1 .. * ] of
  ift$store_terminal_attribute;
```

```

ift$store_terminal_attribute = record
  case key { input } : ift$terminal_attribute_keys of
  = ifc$auto_input_mode =
    auto_input_mode: boolean,
  = ifc$buffer_behind { * } =
    buffer_behind: boolean,
  = ifc$carriage_return_idle =
    carriage_return_idle:
      ift$carriage_return_idle_range,
  = ifc$collector_delimiter { * } =
    collector_delimiter: ift$collector_delimiter,
  = ifc$collector_mode { * } =
    collector_mode: boolean,
  = ifc$echoplex =
    echoplex: boolean,
  = ifc$eoi_string =
    eoi_string: ift$file_mark_string,
  = ifc$eop_string =
    eop_string: ift$file_mark_string,
  = ifc$input_device =
    input_device: ift$input_devices,
  = ifc$line_feed_idle =
    line_feed_idle: ift$line_feed_idle_range,
  = ifc$no_format_effectors =
    no_format_effectors: boolean,
  = ifc>null_attribute =
    ,
  = ifc$output_device =
    output_device: ift$output_devices,
  = ifc$output_translation =
    output_translation: pmt$program_name,
  = ifc$page_wait =
    page_wait: boolean,
  = ifc$prompt_file =
    prompt_file: amt$local_file_name,
  = ifc$prompt_file_id =
    prompt_file_id: amt$file_identifier,
  = ifc$prompt_string =
    prompt_string: ift$prompt_string,
  = ifc$special_editing =
    special_editing: boolean,
  = ifc$transparent_delim_selection =
    transparent_delim_selection:
      ift$transparent_delim_selection,
  = ifc$transparent_end_character =
    transparent_end_character: char,

```

CONSTANT AND TYPE DECLARATIONS

```
= ifc$transparent_end_count =  
  transparent_end_count:  
    ift$transparent_count_range,  
= ifc$transparent_mode =  
  transparent_mode: boolean,  
= ifc$type_ahead =  
  type_ahead: boolean,  
casend,  
recend;  
  
ift$terminal_attribute_keys =  
  (ifc$abort_line_character, ifc$auto_input_mode,  
  ifc$backspace_character, ifc$buffer_behind,  
  ifc$cancel_group_character,  
  ifc$cancel_line_character,  
  ifc$carriage_return_idle, ifc$collector_delimiter,  
  ifc$collector_mode, ifc$echoplex, ifc$eoi_string,  
  ifc$eop_string, ifc$input_device,  
  ifc$line_feed_idle, ifc$network_control_character,  
  ifc$no_format_effectors,  
  ifc>null_attribute, ifc$output_device,  
  ifc$output_flow_control,  
  ifc$output_translation, ifc$page_length,  
  ifc$page_wait, ifc$page_width, ifc$parity,  
  ifc$pause_break_character, ifc$prompt_file,  
  ifc$prompt_file_id, ifc$prompt_string,  
  ifc$special_editing, ifc$terminal_class,  
  ifc$terminal_name, ifc$terminate_break_character,  
  ifc$transparent_delim_selection,  
  ifc$transparent_end_character,  
  ifc$transparent_end_count, ifc$transparent_mode,  
  ifc$type_ahead);  
  
ift$terminal_classes = (ifc$tty_class,  
  ifc$c75x_class, ifc$c721_class, ifc$i2741_class,  
  ifc$tty40_class, ifc$h2000_class, ifc$x364_class,  
  ifc$t4010_class, ifc$hasp_class, ifc$c200out_class,  
  ifc$c711_class, ifc$c714_class, ifc$c73x_class,  
  ifc$i2780_class, ifc$i3780_class);
```

```

ift$terminal_request_attribute = record
  case key { input } : ift$terminal_attribute_keys of
  = ifc$auto_input_mode =
    auto_input_mode: boolean,
  = ifc$carriage_return_idle =
    carriage_return_idle:
      ift$carriage_return_idle_range,
  = ifc$collector_delimiter { * } =
    collector_delimiter: ift$collector_delimiter,
  = ifc$collector_mode { * } =
    collector_mode: boolean,
  = ifc$echoplex =
    echoplex: boolean,
  = ifc$eoi_string =
    eoi_string: ift$file_mark_string,
  = ifc$eop_string =
    eop_string: ift$file_mark_string,
  = ifc$input_device =
    input_device: ift$input_devices,
  = ifc$line_feed_idle =
    line_feed_idle: ift$line_feed_idle_range,
  = ifc$no_format_effectors =
    no_format_effectors: boolean,
  = ifc>null_attribute =
    ,
  = ifc$output_device =
    output_device: ift$output_devices,
  = ifc$output_translation =
    output_translation: pmt$program_name,

  = ifc$page_length =
    page_length: amt$page_length,
  = ifc$page_wait =
    page_wait: boolean,
  = ifc$page_width =
    page_width: amt$page_width,
  = ifc$prompt_file =
    prompt_file: amt$local_file_name,
  = ifc$prompt_file_id =
    prompt_file_id: amt$file_identifier,
  = ifc$prompt_string =
    prompt_string: ift$prompt_string,
  = ifc$special_editing =
    special_editing: boolean,
  = ifc$transparent_delim_selection =
    transparent_delim_selection:
      ift$transparent_delim_selection,

```

CONSTANT AND TYPE DECLARATIONS

```
= ifc$transparent_end_character =  
  transparent_end_character: char,  
= ifc$transparent_end_count =  
  transparent_end_count:  
    ift$transparent_count_range,  
= ifc$transparent_mode =  
  transparent_mode: boolean,  
= ifc$type_ahead =  
  type_ahead: boolean,  
  casend,  
recend;  
  
ift$terminal_request_attributes = array [1 .. * ] of  
  ift$terminal_request_attribute;  
  
ift$transparent_count_range = 1 ..  
  ifc$max_transparent_count_size;  
  
ift$transparent_delim_selection = record  
  enable_end_character: boolean,  
  enable_end_count: boolean,  
  enable_time_out: boolean,  
recend;
```

OS

Constants

```

osc$max_condition = 999999;
osc$max_name_size = 31;
osc$max_page_size = 65536;
osc$max_ring = 15, { Highest ring number (least }
{ privileged). };
osc$max_segment_length = osc$maximum_offset + 1;
osc$max_string_size = 256;
osc$maximum_offset = 7fffffff(16);
osc$maximum_segment = 0fff(16),

osc$min_ring = 1 { Lowest ring number (most }
{ privileged). };
osc$min_page_size = 512;

osc$null_name = '                ';
osc$status_parameter_delimiter = CHR (31) { Unit }
{ Separator } ;

```

Ordinals

```

osc$invalid_ring = 0;
osc$os_ring_1 = 1 { Reserved for Operating System. };
osc$tmtr_ring = 2 { Task Monitor. };
osc$tsrv_ring = 3 { Task services. };
osc$sj_ring_1 = 4 { Reserved for system job. };
osc$sj_ring_2 = 5;
osc$sj_ring_3 = 6;
osc$application_ring_1 = 7 { Reserved for }
{ application subsystems. };
osc$application_ring_2 = 8;
osc$application_ring_3 = 9;
osc$application_ring_4 = 10;
osc$user_ring = 11 { Standard user task. };
osc$user_ring_1 = 12 { Reserved for user...0.S. }
{ requests available. };
osc$user_ring_2 = 13;
osc$user_ring_3 = 14 { Reserved for user...0.S. }
{ requests not available. };
osc$user_ring_4 = 15;

```

Types

```

ost$binary_unique_name = packed record
  processor: pmt$processor,
  year: 1980 .. 2047,
  month: 1 .. 12,
  day: 1 .. 31,
  hour: 0 .. 23,
  minute: 0 .. 59,
  second: 0 .. 59,
  sequence_number: 0 .. 9999999,
recend;

ost$clear_file_space = boolean;

ost$date_time = record
  year: 0 .. 255,
  month: 1 .. 12,
  day: 1 .. 31,
  hour: 0 .. 23,
  minute: 0 .. 59,
  second: 0 .. 59,
  millisecond: 0 .. 999,
recend;

ost$family_name = ost$name;

ost$key_lock = packed record
  global: boolean, { True if value is global key. }
  local: boolean, { True if value is local key. }
  value: ost$key_lock_value, { Key or lock value. }
recend;

ost$key_lock_value = 0 .. 3f(16);

ost$name = string (osc$max_name_size);

ost$name_size = 1 .. osc$max_name_size;

ost$page_size = osc$min_page_size ..
  osc$max_page_size;

ost$pva = packed record
  ring: ost$ring,
  seg: ost$segment,
  offset: ost$segment_offset,
recend;

```



```

ost$relative_pointer = - 7fffffff(16) ..
    7fffffff(16);

ost$ring = osc$invalid_ring ..
    osc$max_ring { Ring number };

ost$segment = 0 ..
    osc$maximum_segment { Segment number };

ost$segment_length = 0 .. osc$max_segment_length;

ost$segment_offset = - (osc$maximum_offset + 1) ..
    osc$maximum_offset;

ost$status = record
    case normal: boolean of
    = FALSE =
        identifier: string (2),
        condition: ost$status_condition,
        text: ost$string,
    casend,
    recend;

ost$status_condition = 0 .. osc$max_condition;

ost$string = record
    size: ost$string_size,
    value: string (osc$max_string_size),
    recend;

ost$string_index = 1 .. osc$max_string_size + 1;

ost$string_size = 0 .. osc$max_string_size;

ost$unique_name = record
    case boolean of
    = TRUE =
        value: ost$name,
    = FALSE =
        dollar_sign: string (1),
        sequence_number: string (7),
        p: string (1),
        processor_model_number: string (1),
        s: string (1),
        processor_serial_number: string (4),
        d: string (1),
        year: string (4),
        month: string (2),
        day: string (2),
        t: string (1),
        hour: string (2),
        minute: string (2),

```

CONSTANT AND TYPE DECLARATIONS

```
    second: string (2),  
    casend,  
recend;  
  
ost$user_identification = record  
    user: ost$user_name,  
    family: ost$family_name,  
recend;  
  
ost$user_name = ost$name;  
  
ost$valid_ring = osc$min_ring ..  
    osc$max_ring { valid ring Number };  
  
ost$wait = (osc$wait, osc$nowait);  
ost$wait_for_lock = (osc$wait_for_lock, osc$nowait_for_lock);
```

PF

Constants

```

pfc$family_name_index = 1;
pfc$master_catalog_name_index =
    pfc$family_name_index + 1;
pfc$maximum_cycle_number = 999;
pfc$maximum_retention = 999;
pfc$minimum_cycle_number = 1;
pfc$minimum_retention = 1;
pfc$subcatalog_name_index =
    pfc$master_catalog_name_index + 1;

```

Types

```

pft$application_info = string (osc$max_name_size);
pft$array_index = 1 .. 7FFFFFFF(16);
pft$change_descriptor = record
    case change_type: pft$change_type of
        = pfc$pf_name_change =
            pfn: pft$name,
        = pfc$password_change =
            password: pft$password,
        = pfc$cycle_number_change =
            cycle_number: pft$cycle_number,
        = pfc$retention_change =
            retention: pft$retention,
        = pfc$log_change =
            log: pft$log,
        = pfc$change_change =
            ,
        casend,
    recend;

```

CONSTANT AND TYPE DECLARATIONS

```
pft$change_list = array [ 1 .. * ] of
  pft$change_descriptor;

pft$change_type = (pfc$pf_name_change,
  pfc$password_change, pfc$cycle_number_change,
  pfc$retention_change, pfc$log_change,
  pfc$charge_change);

pft$cycle_number = pfc$minimum_cycle_number ..
  pfc$maximum_cycle_number;

pft$cycle_options = (pfc$lowest_cycle,
  pfc$highest_cycle, pfc$specific_cycle);

pft$cycle_selector = record
  case cycle_option: pft$cycle_options of
  = pfc$lowest_cycle =
    /
  = pfc$highest_cycle =
    /
  = pfc$specific_cycle =
    cycle_number: pft$cycle_number,
  casend,
  recend;

pft$group = record
  case group_type: pft$group_types of
  = pfc$public =
    /
  = pfc$family =
    family_description: record
      family: ost$family_name,
    recend,
  = pfc$account =
    account_description: record
      family: ost$family_name,
      account: avt$account_name,
    recend,
  = pfc$project =
    project_description: record
      family: ost$family_name,
      account: avt$account_name,
      project: avt$project_name,
    recend,
  = pfc$user =
    user_description: record
      family: ost$family_name,
      user: ost$user_name,
    recend,
```

```

= pfc$user_account =
  user_account_description: record
    family: ost$family_name,
    account: avt$saccount_name,
    user: ost$user_name,
  recend,
= pfc$member =
  member_description: record
    family: ost$family_name,
    account: avt$saccount_name,
    project: avt$project_name,
    user: ost$user_name,
  recend,
casend,
recend;

pft$group_types = (pfc$public, pfc$family,
  pfc$saccount, pfc$project, pfc$user,
  pfc$user_account, pfc$member),

pft$log = (pfc$log, pfc$no_log);

pft$name = ost$name;

pft$password = pft$name;

pft$path = array [ 1 .. * ] of pft$name;

pft$permit_options = (pfc$read, pfc$shorten,
  pfc$append, pfc$modify, pfc$execute, pfc$cycle,
  pfc$control);

pft$permit_selections = set of pft$permit_options;

pft$retention = pfc$minimum_retention ..
  pfc$maximum_retention;

pft$share_options = pfc$read .. pfc$execute;

pft$share_selections = set of pft$share_options;

pft$share_requirements = set of pft$share_options;

pft$usage_options = pfc$read .. pfc$execute;

pft$usage_selections = set of pft$usage_options;

pft$wait = (pfc$wait, pfc$no_wait);

```

PM

Types

```
pmt$cpu_model_number = (pmc$cpu_model_p1,  
  pmc$cpu_model_p2, pmc$cpu_model_p3,  
  pmc$cpu_model_p4);
```

```
pmt$cpu_serial_number = 0 .. 0ffff(16);
```

```
pmt$processor = record  
  serial_number: pmt$cpu_serial_number,  
  model_number: pmt$cpu_model_number,  
  recend;
```

```
pmt$processor_attributes = record  
  model_number: pmt$cpu_model_number,  
  serial_number: pmt$cpu_serial_number,  
  page_size: ost$page_size,  
  recend;
```

```
pmt$program_name = ost$name;
```

RM

Constants

```
rmc$external_vsn_size = 6;  
rmc$max_ecc_resource_management = 249999;  
rmc$min_ecc_resource_management = 240000;  
rmc$recorded_vsn_size = 6;  
rmc$resource_management_id = 'RM';
```

Types

```
rmt$density = (rmt$200, rmt$556, rmt$800, rmt$1600,
  rmt$6250);
```

```
rmt$device_class = (rmt$mass_storage_device,
  rmt$magnetic_tape_device, rmt$terminal_device,
  rmt$null_device);
```

```
rmt$external_vsn = string (rmt$external_vsn_size);
```

```
rmt$recorded_vsn = string (rmt$recorded_vsn_size);
```

```
rmt$release_tape_request = record
  tape_class: rmt$tape_class,
  nt9$800_count: integer,
  nt9$1600_count: integer,
  nt9$6250_count: integer,
  recend;
```

```
rmt$reserve_tape_request = record
  tape_class: rmt$tape_class,
  nt9$800_count: integer,
  nt9$1600_count: integer,
  nt9$6250_count: integer,
  recend;
```

```
rmt$tape_class = (rmt$mt7, rmt$mt9);
```

```
rmt$volume_descriptor = record
  recorded_vsn: rmt$recorded_vsn,
  external_vsn: rmt$external_vsn,
  recend;
```

```
rmt$volume_list = array [ * ] of
  rmt$volume_descriptor;
```

```
rmt$write_ring = (rmt$write_ring,
  rmt$no_write_ring);
```



A file access procedure (FAP) is a procedure that acts as an interface between calls to file interface procedures and the actual file processing by the procedures.

If a file has a FAP associated with it, the system passes file interface calls from a task to the FAP, instead of processing the call immediately itself. Table D-1 lists the procedure calls passed to a FAP. In general, the calls passed to the FAP are the file interface calls issued during an instance of open. The calls passed do not include the calls that can be issued only within a FAP or the calls generally issued only before or after an instance of open. Each procedure call passed to a FAP specifies a file identifier parameter.

The FAP determines the processing of a call passed to it. The FAP can call the system to process the call or it can simulate system processing. All system simulation must be compatible with actual system processing.

The user writes a FAP to provide a service not supported by the system. The following are possible services a FAP could provide:

- Data encryption and decryption.
- I/O modeling.
- Data conversion.
- Data compression.
- Logging of changes or accesses to the file.
- Trapping I/O operations during debugging.

Table D-1. Calls Passed to a FAP

Procedure Call	Call Block Identifier
AMP\$ABANDON_KEY_DEFINITIONS	AMC\$ABANDON_KEY_DEFINITIONS
AMP\$ABORT_FILE_PARCEL	AMC\$ABORT_FILE_PARCEL
AMP\$APPLY_KEY_DEFINITIONS	AMC\$APPLY_KEY_DEFINITIONS
AMP\$BEGIN_FILE_PARCEL	AMC\$BEGIN_FILE_PARCEL
AMP\$CHECK_NOWAIT_REQUEST	AMC\$CHECK_NOWAIT_REQUEST
AMP\$CLOSE	AMC\$CLOSE_REQ
AMP\$COMMIT_FILE_PARCEL	AMC\$COMMIT_FILE_PARCEL
AMP\$CREATE_KEY_DEFINITIONS	AMC\$CREATE_KEY_DEFINITIONS
AMP\$CREATE_NESTED_FILE	AMC\$CREATE_NESTED_FILE
AMP\$DELETE_KEY	AMC\$DELETE_KEY_REQ
AMP\$DELETE_KEY_DEFINITIONS	AMC\$DELETE_KEY_DEFINITIONS
AMP\$DELETE_NESTED_FILE	AMC\$DELETE_NESTED_FILE
AMP\$FETCH	AMC\$FETCH_REQ
AMP\$FETCH_ACCESS_INFORMATION	AMC\$FETCH_ACCESS_INFORMATION_REQ
AMP\$FLUSH	AMC\$FLUSH_REQ
AMP\$GET_DIRECT	AMC\$GET_DIRECT_REQ
AMP\$GET_KEY	AMC\$GET_KEY_REQ
AMP\$GET_KEY_DEFINITIONS	AMC\$GET_KEY_DEFINITIONS
AMP\$GET_LOCK_KEYED_DEFINITIONS	AMC\$GET_LOCK_KEYED_DEFINITIONS
AMP\$GET_LOCK_KEYED_RECORD	AMC\$GET_LOCK_KEYED_RECORD
AMP\$GET_LOCK_NEXT_KEYED_RECORD	AMC\$GET_LOCK_NEXT_KEYED_RECORD
AMP\$GET_NEXT	AMC\$GET_NEXT_REQ
AMP\$GET_NEXT_KEY	AMC\$GET_NEXT_KEY_REQ
AMP\$GET_NEXT_PRIMARY_KEY_LIST	AMC\$GET_NEXT_PRIMARY_KEY_LIST
AMP\$GET_PARTIAL	AMC\$GET_PARTIAL_REQ
AMP\$GET_PRIMARY_KEY_COUNT	AMC\$GET_PRIMARY_KEY_COUNT

(Continued)

Table D-1. Calls Passed to a FAP *(Continued)*

Procedure Call	Call Block Identifier
AMP\$GET_SEGMENT_POINTER	AMC\$GET_SEGMENT_POINTER_REQ
AMP\$GET_SPACE_USED_FOR_KEY	AMC\$GET_SPACE_USED_FOR_KEY
AMP\$OPEN	AMC\$OPEN_REQ
AMP\$PUT_DIRECT	AMC\$PUT_DIRECT_REQ
AMP\$PUT_KEY	AMC\$PUT_KEY_REQ
AMP\$PUT_NEXT	AMC\$PUT_NEXT_REQ
AMP\$PUT_PARTIAL	AMC\$PUT_PARTIAL_REQ
AMP\$PUTREP	AMC\$PUTREP_REQ
AMP\$REPLACE_KEY	AMC\$REPLACE_KEY_REQ
AMP\$REWIND	AMC\$REWIND_REQ
AMP\$SEEK_DIRECT	AMC\$SEEK_DIRECT_REQ
AMP\$SELECT_KEY	AMC\$SELECT_KEY
AMP\$SELECT_NESTED_FILE	AMC\$SELECT_NESTED_FILE
AMP\$SEPARATE_KEY_GROUPS	AMC\$SEPARATE_KEY_GROUPS
AMP\$SET_SEGMENT_EOI	AMC\$SET_SEGMENT_EOI_REQ
AMP\$SET_SEGMENT_POSITION	AMC\$SET_SEGMENT_POSITION_REQ
AMP\$SKIP	AMC\$SKIP_REQ
AMP\$START	AMC\$START_REQ
AMP\$STORE	AMC\$STORE_REQ
AMP\$UNLOCK_KEY	AMC\$UNLOCK_KEY
AMP\$WRITE_END_PARTITION	AMC\$WRITE_END_PARTITION_REQ
AMP\$WRITE_TAPE_MARK	AMC\$WRITE_TAPE_MARK_REQ
IFP\$FETCH_TERMINAL	IFC\$FETCH_TERMINAL_REQ
IFP\$STORE_TERMINAL	IFC\$STORE_TERMINAL_REQ

File Access Procedure Attribute

The `file_access_procedure` attribute names the FAP for the file. A FAP is optional; the attribute has no default value.

To associate a FAP with a file, you specify the FAP name as the `file_access_procedure` attribute value for a new file before the file is opened. The means of defining attribute values for a new file are described in chapter 6.

If the `file_access_procedure` attribute has a value when the file is first opened, the attribute value is preserved with the file. Thereafter, a FAP remains associated with the file for the lifetime of the file. You can specify or change the FAP associated with an old file (a file that has been opened) with a `CHANGE_FILE_ATTRIBUTES` command.

NOTE

Although you can change the FAP attribute value of an old file, you cannot clear the FAP attribute value. Once a FAP is associated with the file, the file must continue to have a FAP associated with it, although it need not be the original FAP. After a new file is opened, the FAP name is preserved with the file and a task cannot open the file unless the system can load the FAP with the task.

FAP Loading

Each time an `AMP$OPEN` call attempts to open a file, it attempts to load the FAP named by the `file_access_procedure` file attribute (if the FAP is not already loaded for the task). The FAP must either be named in the current task object library list or be an entry point in the task itself. If the FAP cannot be loaded, the attempt to open the file fails, and the `AMP$OPEN` call returns abnormal status (`AME$UNABLE_TO_LOAD_FAP`).

A FAP does not acquire additional privilege by being associated with a file. It has the privilege of the object file or object library from which it is loaded qualified by the ring of the caller of `AMP$OPEN`.

Assigning a FAP to a file does not in itself give the FAP permission to access the file. A FAP must have sufficient ring privilege to access the file in some way or it cannot be loaded when the file is accessed. However, if the type of access granted to the task is not the access required to perform the function passed to the FAP, the FAP should either simulate that type of file access or return abnormal status.

FAP Declaration

After loading the FAP, the system calls the FAP for each file interface call it receives from the task. When it calls the FAP, it passes information from the file interface call.

When the system calls a FAP, it passes the file identifier and call block from the file interface call; it also passes a layer number and a status variable.

The FAP declaration must have the following form:

```
PROCEDURE [XDCL] name
  (file_identifier: amt$file_identifier;
   call_block: amt$call_block;
   layer_number: amt$fap_layer_number;
   VAR status: ost$status);
```

file_identifier

File identifier specified on the file interface call.

call_block

Actual parameters specified on the system call. The call_block also contains the call identifier as listed in table D-1.

layer_number

Layer number of the FAP (integer from 0 through 15).

NOTE

The FAP must not modify the layer number passed to it. If the FAP calls AMP\$ACCESS_METHOD, it must pass the layer number, unmodified, to the procedure.

Although the layer_number type has the range 0 through 15, only one FAP layer (the uppermost) is available to the user. All other FAP layers are available only to the system.

status

Status record specified on the file interface call.

Call Block

Within the call_block, the system passes the actual parameters from the file interface call to the FAP. For each parameter in the parameter list, the call_block contains either the parameter value or a pointer to the parameter value.

The call_block type, AMT\$CALL_BLOCK, as listed in appendix C, is a variant record. The key field is the operation field that contains one of the call identifiers listed in table D-1. As listed in appendix C, the call identifier determines the other fields in the record. The record contains one field for each parameter on the file interface call (except the file identifier and status parameters). For more information about the parameter values passed, see the individual call description.

FAP Processing

NOS/VE provides the following special system procedures for use by a FAP:

- **AMP\$VALIDATE_CALLER_PRIVILEGE:**
Ensures that the caller has the privilege required for the processing performed.
- **AMP\$ADD_TO_FILE_DESCRIPTION:**
Allows the FAP to specify a file attribute value when opening a new file if the user has not explicitly defined the attribute.
- **AMP\$ACCESS_METHOD:**
Allows the FAP to access the file with which it is associated.
- **AMP\$STORE_FAP_POINTER** and **AMP\$FETCH_FAP_POINTER:**
Allows the FAP to request and use a pointer to its own data structure.
- **AMP\$SET_FILE_INSTANCE_ABNORMAL:**
Allows the FAP to set abnormal status for an instance of open.

FAP Security

When writing a FAP that executes in a lower (more secure) ring than the file accessor, you must ensure that the FAP protects file security. It must check that the caller that requests an operation has the required privilege.

For example, a task with read-only access to a file could open a file for read access and subsequently request a write operation. If the FAP associated with the file has write access to the file, it must check that its caller also has the necessary privilege before performing the operation.

The FAP checks the privilege of a caller by calling **AMP\$VALIDATE_CALLER_PRIVILEGE**. If the call returns abnormal status, the FAP must terminate and return the abnormal status to its caller.

AMP\$VALIDATE_CALLER_PRIVILEGE

- Purpose** Ensures that the caller of a FAP is authorized to perform the requested operation. It also returns a pointer to the FAP data structure.
- Format** **AMP\$VALIDATE_CALLER_PRIVILEGE** (**file_** **identifier**, **layer_number**, **required_write_privilege**, **caller_ring_number**, **structure_pointer**, **status**)
- Parameters**
- file_identifier**: amt\$file_identifier;
File identifier passed to the FAP when it was called.
- layer_number**: amt\$fap_layer_number;
Layer number passed to the FAP when it was called.
- required_write_privilege**: pft\$usage_selections;
Set of one or more access modes that must have been included in the access privileges for this instance of open of the file. Required for write operations; ignored for other operations.
- PFC\$SHORTEN**
Shorten access.
- PFC\$APPEND**
Append access.
- PFC\$MODIFY**
Modify access.
- caller_ring_number**: ost\$ring;
Ring of the caller of the FAP. To get the ring number, use the #CALLER_ID procedure described in the CYBIL Language Definition manual and reference the ring field of the record the function returns.
- structure_pointer**: VAR of ^cell;
Pointer to FAP data structure.
- status**: VAR of ost\$status;
Status variable. The process identifier returned is AMC\$ACCESS_METHOD_ID.

Condition Identifiers

ame\$improper_access_attempt
 ame\$improper_fap_operation
 ame\$improper_file_id
 ame\$null_set_specified
 ame\$ring_validation_error

Remarks

- Use of this call is mandatory when the caller of a FAP could have less privilege than the FAP. If the caller does not have the privilege required to perform the operation specified in the call block passed to the FAP, the AMP\$VALIDATE_CALLER_PRIVILEGE call returns either AME\$RING_VALIDATION_ERROR or AME\$IMPROPER_ACCESS_ATTEMPT status.
- For write operations, the call must specify the privilege required to perform the operation. This allows the FAP to determine the required access modes for each type of write operation.
- For example, an AMP\$PUT_NEXT operation for a sequential file requires either append or shorten access. The access required depends on the current byte address of the file (returned by an AMP\$FETCH_ACCESS_INFORMATION call). If the address is before the end of the file, it requires shorten access. If it is at the end of the file, it requires append access.
- The FAP data structure pointer returned is NIL if a pointer has not previously been stored by an AMP\$STORE_FAP_POINTER call.

File Attribute Specification

While a new file is opened, the FAP can specify additional file attribute values with an AMP\$ADD_TO_FILE_DESCRIPTION call. The call is only valid for the first instance of open of the file.

The file attributes specified on the call must be attributes that have not been explicitly defined; that is, the attribute value is as yet undefined or a default value has been supplied by the system. If the call attempts to define an explicitly defined file attribute, an abnormal status is returned (AME\$ATTRIB_ALREADY_DEFINED).

AMP\$ADD_TO_FILE_DESCRIPTION

Purpose Defines file attribute values of a new file within a FAP.

NOTE

The call is valid only within a FAP and only when the system calls the FAP due to an AMP\$OPEN call for a new file. The only file attributes that can be defined by the call are those that are undefined or defined by default attribute values, that is, those that have not been explicitly defined by the task.

Format AMP\$ADD_TO_FILE_DESCRIPTION (**file_identifier**, **file_attributes**, **status**)

Parameters **file_identifier**: amt\$file_identifier;

File identifier passed to FAP.

file_attributes: amt\$add_to_attributes;

Attribute array containing one record for each attribute to be defined. You must specify an attribute identifier and an attribute value in each record. The valid attributes are listed in table 6-1.

status: VAR of ost\$status;

Status record specified on the access method call. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers

- ame\$attrib_already_defined
- ame\$improper_file_attr_key
- ame\$improper_file_attr_value
- ame\$improper_file_id
- ame\$not_open_new
- ame\$ring_validation_error

Remarks

- The FAP can determine the undefined attributes by calling AMP\$FETCH. If the attribute is undefined, AMC\$UNDEFINED is returned as the source of the file attribute value. See table 6-2 for a list of file attribute sources.
- For a file with sequential or byte addressable file organization, the call cannot change the max_block_length, max_record_length, block_type, or record_type attributes.
- Although more than one FAP layer can specify a value for a file attribute, the first definition of an attribute overrides all others.

FAP System Calls

A FAP can pass a file interface call it receives to the system with an AMP\$ACCESS_METHOD call. The parameter list for the AMP\$ACCESS_METHOD call is the same as the parameter list passed to the FAP for a call.

When using the AMP\$ACCESS_METHOD interface, the FAP can only reference the instance of open specified on the call to the FAP. The FAP can access other files using calls other than AMP\$ACCESS_METHOD.

When the system calls the FAP for a file, the task has already issued the AMP\$OPEN call for the file. The FAP must pass the AMP\$OPEN call block to the system using an AMP\$ACCESS_METHOD call before issuing other AMP\$ACCESS_METHOD calls. The AMP\$ACCESS_METHOD call passing the AMP\$OPEN call block provides other system FAP layers with the open information.

Because the FAP passes the AMP\$OPEN call, it must also pass an AMP\$CLOSE call. If the FAP does not pass the AMP\$CLOSE call, the end-user program cannot gain access to any abnormal status which may have been generated as a result of close processing by other system FAP layers. In this case, the system puts the abnormal status into the job and system logs.

If the FAP emulates system processing rather than passing the requests to the access method, it must execute each request in a manner functionally compatible with the access method it replaces. This includes initializing the status record and other actual parameters, where applicable.

AMP\$ACCESS_METHOD

Purpose	Transfers control to the system from a FAP. A FAP uses this call to pass a call it receives to the system or to issue its own system requests.
Format	AMP\$ACCESS_METHOD (file_identifier , call_block , fap_layer_number , status)
Parameters	<p>file_identifier: amt\$file_identifier; File identifier passed from the system call.</p> <p>call_block: amt\$call_block; Actual parameters for the access method call (type AMT\$CALL_BLOCK as defined in appendix C).</p> <p>layer_number: amt\$fap_layer_number; FAP layer number passed to the FAP when it was called (integer from 0 through 15).</p> <p>status: VAR of ost\$status; Status record specified on the access method call. The process identifier is AMC\$ACCESS_METHOD_ID.</p>
Condition Identifiers	<p>ame\$improper_fap_operation</p> <p>ame\$improper_file_id</p> <p>ame\$improper_layer_number</p> <p>ame\$ring_validation_error</p>

- Remarks**
- To pass a system request to the system, the FAP passes the parameter list it received when it was called as the parameter list on the AMP\$ACCESS_METHOD call. That is, it specifies the formal parameters from its declaration statement as the actual parameter list of the AMP\$ACCESS_METHOD call (see the FAP example).
 - To issue its own system request, the FAP must initialize its own call_block values for the call. The layer_number value must be the same passed to the FAP when it was called; the same layer_number value is used on all AMP\$ACCESS_METHOD calls from the FAP.

FAP Data Structure

When more than one file opened within a task has the same FAP associated with it, each file shares the same copy of the FAP and the same static variable space defined within the FAP. If a data structure should be associated with only one instance of open, the FAP should dynamically allocate the structure for each instance of open.

To associate a dynamically allocated structure with each instance of open, the FAP should use the AMP\$STORE_FAP_POINTER and AMP\$GET_FAP_POINTER calls as follows:

1. If the FAP is called as the result of an AMP\$OPEN call, it should allocate the data structure and then call AMP\$STORE_FAP_POINTER to store a pointer to the structure.
2. Later, when the FAP is called due to other file interface calls that reference the instance of open, the FAP can call AMP\$FETCH_FAP_POINTER to get a pointer to the structure associated with the instance of open. It can then reference the structure using the pointer.

The FAP example in this section illustrates use of a FAP data structure to maintain a count.

AMP\$STORE_FAP_POINTER

Purpose Stores a pointer to a data structure associated with an instance of open.

Format AMP\$STORE_FAP_POINTER (**file_identifier**, **layer_number**, **structure_pointer**, **status**)

Parameters **file_identifier**: amt\$file_identifier;
File identifier passed to the FAP.

layer_number: amt\$fap_layer_number;
FAP layer number passed to the FAP when it was called (integer from 0 through 15).

structure_pointer: ^cell;
Pointer to a data structure (^cell).

status: VAR of ost\$status;
Status record. The process identifier is AMC\$ACCESS_METHOD_ID.

Condition Identifiers ame\$improper_file_id
ame\$redundant_structure_pointer
ame\$ring_validation_error

Remarks

- Using this procedure, the FAP can allocate a different data structure for each instance of open.
- If the structure pointer has already been stored for the specified FAP layer number, the call returns the AME\$REDUNDANT_STRUCTURE_POINTER exception condition.

AMP\$FETCH_FAP_POINTER

Purpose	Returns a pointer to the data structure owned by the FAP.
Format	AMP\$FETCH_FAP_POINTER (file_identifier , layer_number , structure_pointer , status)
Parameters	file_identifier : amt\$file_identifier; File identifier passed to the FAP. layer_number : amt\$fap_layer_number; FAP layer number passed to the FAP when it was called (integer from 0 through 15). structure_pointer : VAR of ^cell; Pointer to the data structure owned by the FAP (^cell). status : VAR of ost\$status; Status record. The process identifier is AMC\$ACCESS_METHOD_ID.
Condition Identifiers	ame\$improper_file_id ame\$nil_structure_pointer ame\$ring_validation_error
Remarks	<ul style="list-style-type: none">• The data structure is unique to the instance of open of the file.• The pointer must be stored by an AMP\$STORE_FAP_POINTER call within the FAP. If the FAP pointer is NIL, the procedure returns abnormal status to the FAP.

FAP Example

The following is an example of a FAP that counts the number of AMP\$GET_NEXT calls to the file issued during an instance of open. The count is kept until the file is closed. All calls are passed to the access method for processing.

```
MODULE fap_example;
```

```
*copyc amp$store_fap_pointer
*copyc amp$fetch_fap_pointer
*copyc amp$access_method
```

```
PROCEDURE [XDCL] fap_example
  (file_id: amt$file_identifier;
   call_block: amt$call_block;
   layer: amt$fap_layer_number;
   VAR status: ost$status);
```

```
VAR
  access_count: ^integer,
  fap_stat: ost$status;
```

```
{ In response to an AMP$OPEN call, the procedure }
{ allocates an integer variable in the CYBIL    }
{ heap, stores the pointer to the variable     }
{ (access_count), and initializes the cell to 0. }
```

```
IF call_block.operation = amc$open_req THEN
  ALLOCATE access_count;
```

```
AMP$STORE_FAP_POINTER (file_id, layer,
  access_count, fap_stat);
access_count^ := 0;
```

AMP\$FETCH_FAP_POINTER

```
{ In response to a call other than AMP$OPEN, the }  
{ procedure fetches the pointer to the access }  
{ count. If the call is an AMP$GET_NEXT call, }  
{ the procedure increments the count. If the }  
{ call is an AMP$CLOSE call, the procedure frees }  
{ the access count variable. }  
}
```

ELSE

```
AMP$FETCH_FAP_POINTER (file_id, layer,  
    access_count, fap_stat);
```

```
CASE call_block.operation OF  
= amc$get_next_req =  
    access_count^ := access_count^ + 1;  
= amc$close_req =  
    FREE access_count;
```

ELSE

;

CASEND;

IFEND;

```
{ Each call block is passed to the access method }  
{ for processing. }  
}
```

```
AMP$ACCESS_METHOD (file_id, call_block, layer,  
    status);
```

PROCEND fap_example;

MODEND fap_example;

FAP Error Reporting

If, after the FAP receives the AMP\$OPEN call, it detects an exception condition, the FAP must return with abnormal status to AMP\$OPEN. The AMP\$OPEN interface then detects the exception condition, sends a close operation to the FAP, and returns the abnormal status to the program. In this manner, all FAP layers receive the close operation.

To initialize the status record to indicate an abnormal condition, the FAP can call AMP\$SET_FILE_INSTANCE_ABNORMAL. The AMP\$SET_FILE_INSTANCE_ABNORMAL procedure generates a status record, recording the following status parameters in the text field of the status record:

- P1: Local file name.
- P2: Name of the access method request that detected the condition.
- P3: Access level.
- P4: File organization.
- P5: Record type.
- P6: Block type.
- P7: Information reserved for internal use by the access method.
- P8: Text string specified on the AMP\$SET_FILE_INSTANCE_ABNORMAL call.

By convention, message templates use the Pn notation to indicate the status parameters inserted in the template. The applicable message templates are in deck AME\$EXCEPTION_CONDITION_CODES on file \$SYSTEM.OSF\$PROGRAM_INTERFACE_LIBRARY and deck AAE\$EXCEPTION_CONDITION_CODES on file \$SYSTEM.COMMON.PSF\$EXTERNAL_INTERFACE_SOURCE.

NOTE

The text parameter on the AMP\$SET_FILE_INSTANCE_ABNORMAL call can specify a null string unless the message template for the specified condition code indicates that the P8 status parameter is inserted in the template.

AMP\$SET_FILE_INSTANCE_ABNORMAL

Purpose	Sets abnormal status for an instance of open.
Format	AMP\$SET_FILE_INSTANCE_ABNORMAL (file_ identifier, exception_ condition, request_ code, text, status)
Parameters	<p>file_ identifier: amt\$file_ identifier; File identifier passed to the FAP.</p> <p>exception_ condition: ost\$status_ condition; Condition code for the exception condition. A condition code can be specified with its condition identifier. The condition codes and condition identifiers are listed in the Diagnostic Messages for NOS/VE manual.</p> <p>request_ code: amt\$last_ operation; Code specifying the access method request that detected the condition (type AMT\$LAST_ OPERATION as listed in the last_ operation attribute description in chapter 6).</p> <p>text: string (*); String to be appended to the text field of the status record as the eighth status parameter.</p> <p>The string is delimited by the OSC\$STATUS_ PARAMETER_ DELIMITER character. If no text string is to be appended, the call must specify a null string (two consecutive delimiters).</p> <p>status: VAR of ost\$status; Status record to be initialized.</p>
Condition Identifier	None.

Collation Tables for Indexed Sequential Files

One of the key types available for use with indexed sequential files (as described in chapter 10) is collated keys. The order in which collated keys are sorted is determined by a collation table. If you specify the `KEY_TYPE` attribute for the file as `AMC$COLLATED_KEY`, you must supply an explicit collation table; there is no system-supplied default collation table.

You specify a collation table by specifying its name as the `COLLATE_TABLE_NAME` attribute value before the file is first opened. The collation table name can be the name of a system-defined table or a user-defined table. The names of the system-defined tables are listed in the next section; the means of creating your own collation table are described later in this appendix.

The collation table you assign a new file can be the collation table of an old file. You fetch the collation table of the old file using an `AMP$FETCH` call. Before calling `AMP$FETCH`, you define a variable of type `AMT$COLLATE_TABLE` with attribute `XDCL` and store a pointer to the variable as the `COLLATE_TABLE` attribute in the `AMP$FETCH` attribute record. `AMP$FETCH` copies the collation table to the variable you defined. You can then specify the same variable name as the `COLLATE_TABLE_NAME` attribute for the new file.

The tables in this appendix list collating sequences, not character sets. A character set shows the codes used for internal representation of character data. For `NOS/VE`, there is only one character set: `ASCII`. A `NOS/VE` collating sequence is an ordering of the character codes in the `ASCII` character set.

Certain of the collating sequences listed in this appendix use the names of other character sets, such as `CDC` display code, in their names. These collating sequences order character data the same as a system using the other character set would order the data. For example, data ordered using a display code collating sequence on `NOS/VE` is ordered the same as a `NOS` system would order display code data.

System-Defined Collation Tables

The collating sequences of the predefined collation tables are listed in tables E-1 through E-11.

Several of the predefined collation tables have two variants, FOLDED and STRICT. The variants FOLDED and STRICT indicate different mapping of the characters not in the 63 or 64 characters of the original CYBER 170 collating sequence. A strict mapping maps all characters not in the original 64- or 63-character set to the ordinal for the space character. A folded mapping maps some characters into ordinals of the original characters and the others into the ordinal value for the space character as shown in the listing of the collating sequence.

- OSV\$ASCII6_FOLDED and OSV\$ASCII6_STRICT:
CYBER 170 FTN 5 default collating sequence.
- OSV\$COBOL6_FOLDED and OSV\$COBOL6_STRICT:
CYBER 170 COBOL 5 default collating sequence.
- OSV\$DISPLAY63_FOLDED and OSV\$DISPLAY63_STRICT:
CYBER 170 63-character display code collating sequence.
- OSV\$DISPLAY64_FOLDED and OSV\$DISPLAY64_STRICT:
CYBER 170 64-character display code collating sequence.
- OSV\$EBCDIC:
Full EBCDIC collation sequence.
- OSV\$EBCDIC6_FOLDED and OSV\$EBCDIC6_STRICT:
EBCDIC 6-bit subset supported by CYBER 170 COBOL 5 and SORT 5.

Creating Your Own Collation Table

CYBIL creates a collation table whose name is specified by the `COLLATE_TABLE_NAME` attribute. The collation table is 256 bytes which represent collating weights associated with the ASCII character set.

These steps are required to create your own collation table and assign it to a new file:

1. Define a variable of type `AMT$COLLATE_TABLE` with attribute `XDCL`. For example:

```
VAR
    reverse_ascii: [XDCL] amt$collate_table;
```

2. Specify `AMC$COLLATED_KEY` as the `KEY_TYPE` attribute value and the variable name as the `COLLATE_TABLE_NAME` attribute value for the file. For example, these lines would be included when initializing the file attribute record:

```
[amc$key_type, amc$collated_key],
[amc$collate_table_name, 'REVERSE_ASCII'],
```

Note that all letters in the collation table name are in uppercase (required to load the table).

3. Initialize the variable.

The `AMT$COLLATE_TABLE` variable is an array of 256 integers, one for each character in the ASCII character set. Each array element is referenced by its corresponding character (for example, if the array name is `COLLATE_TABLE`, `COLLATE_TABLE['a']` references the array element for character a).

To initialize the collation table, you assign an integer to each element of the array. The integer is the collating weight for the corresponding ASCII character. For example, assigning the value 0 to `COLLATE_TABLE['a']` assigns collating weight 0 to character a.

If you assign the consecutive integers 0 through 255 to the array elements, the collation table defines the standard ASCII collating sequence as listed in appendix B. To define a collation table that collates in reverse order from the standard ASCII collation sequence, you would assign the integers 0 through 255 in reverse order to the array elements. For example:

```
FOR i:=0 TO 255 DO
    reverse_ascii[$CHAR(i)] := 255-i;
FOREND;
```

4. Ensure that the loader can find the variable when the file is opened. The variable must be defined as an XDCL variable in a module that is part of the program or can be loaded during program execution. (To read about module loading, see the SCL Object Code Management manual.)
5. Open the new file. The system stores the collation table in the file label.

Assigning the Same Collating Weight to More Than One Character

Your collation table can assign the same collating weight to more than one character. Characters with the same collating weight are considered to be equal when compared during sorting. If all characters have the same collating weight, a sort would perform no reordering because all characters would be considered as equal.

Consider the requirement that a collation table consider each pair of uppercase and lowercase letters as equal; that is, the collation would be case insensitive. To create such a collation table, you would want to keep the standard ASCII collating sequence except that lowercase letters are collated the same as the corresponding uppercase letters. Assuming an AMT\$COLLATE_TABLE variable names NO_LOWER_CASE has been declared, the collation table initialization could be performed in two steps:

1. The collation table variable is initialized to the standard ASCII collating sequence. (The character "i" is declared as an integer variable.)

```
FOR i := 0 TO 255 DO
  no_lower_case[$CHAR(i)] := i;
FOREND;
```

2. The collating weights for the lowercase letters are changed to match the collating weights for the uppercase letters. (UPPER and LOWER are declared as character variables.)

```
upper := 'A';
FOR lower := 'a' TO 'z' DO
  no_lower_case[lower] := $INTEGER(upper);
  upper := SUCC(upper);
FOREND;
```


Assigning a Collation Table Within a File Access Procedure

A FAP can specify a collation table when the following conditions are true:

- No collation table has been specified for the file.
- The FAP has been called as the result of the first AMP\$OPEN call for the file.

A FAP specifies a collation table using the `collate_table` attribute, not the `collate_table_name` attribute. It specifies the `collate_table` attribute value using an AMP\$ADD_TO_FILE_DESCRIPTION call.

The AMP\$ADD_TO_FILE_DESCRIPTION call specifies a pointer to a collation table as the value of the `collte_table` attribute. When it opens the file, the system stores the collation table in the file label.

COLLATION TABLES FOR INDEXED SEQUENTIAL FILES

Table E-1. OSV\$ASCII6_FOLDED Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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
46	4E,6E	N,n	Uppercase N, lowercase n

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-1. OSV\$ASCII6_FOLDED Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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	ˆ , ˜	Circumflex, tilde
63	5F	_	Underline

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-2. OSV\$ASCII6_STRICT Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-2. OSV\$ASCII6_STRICT Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-3. OSV\$COBOL6_FOLDED Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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
46	54,74	T,t	Uppercase T, lowercase t

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-3. OSV\$COBOL6_FOLDED Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-4. OSV\$COBOL6_STRICT Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-4. OSVS\$COBOL6_STRICT Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-5. OSV\$DISPLAY63_FOLDED Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-5. OSV\$DISPLAY63_FOLDED Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
45	2C	,	Comma
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

† Any ASCII code not listed here (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-6. OSV\$DISPLAY63_STRICT Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-6. OSV\$DISPLAY63_STRICT Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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
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

† Any ASCII code not listed here (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-7. OSV\$DISPLAY64_FOLDED Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-7. OSV\$DISPLAY64_FOLDED Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-8. OSV\$DISPLAY64—STRICT Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-8. OSVS\$DISPLAY64_STRICT Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-9. OSV\$EBCDIC Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code	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

(Continued)

Table E-9. OSV\$EBCDIC Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code	Graphic or Mnemonic	Name or Meaning
051	93	—	Unassigned
052	94	—	Unassigned
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

(Continued)

Table E-9. OSV\$EBCDIC Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code	Graphic or Mnemonic	Name or Meaning
101	B5	—	Unassigned
102	B6	—	Unassigned
103	B7	—	Unassigned
104	B8	—	Unassigned
105	B9	—	Unassigned
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

(Continued)

Table E-9. OSV\$EBCDIC Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table E-9. OSV\$EBCDIC Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code	Graphic or Mnemonic	Name or Meaning
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
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 E-10. OSV\$EBCDIC6_FOLDED Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code†	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
46	53,73	S,s	Uppercase S, lowercase s
47	54,74	T,t	Uppercase T, lowercase t

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-10. OSV\$EBCDIC6_FOLDED Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table E-11. OSV\$EBCDIC6_STRICT Collating Sequence

Collating Sequence Position	Hexadecimal ASCII Code	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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

(Continued)

Table E-11. OSV\$EBCDIC6_STRICT Collating Sequence (Continued)

Collating Sequence Position	Hexadecimal ASCII Code†	Graphic or Mnemonic	Name or Meaning
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
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

† Any ASCII code not listed here (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).



Common Procedures

F

Each CYBIL procedure call specified a status variable in which the completion status of the call is returned. After the call, the program checks the status returned. The examples in chapter 10 call the `p#inspect_status_variable` procedure to check the status after each call. Use of the `p#inspect_status_variable` procedure also requires calls to `p#start_report_generation` and `p#stop_report_generation` at the beginning and end of the program, respectively.

The program examples in chapter 10 copy a deck named `COMPROC` to include the common procedures in the program. The following is a listing of the text stored in deck `COMPROC`.

```
CONST
  line_length = 137;

SECTION s#dry_storage_area : READ;
VAR
  error_count           : [STATIC] integer,
  report_file_name     : [STATIC, READ, s#dry_storage_area]
                        amt$local_file_name := '$output',
  report_file_identifier : [STATIC] amt$file_identifier,
  text_index           : [STATIC] 1 .. line_length+1,
  text_line            : [STATIC] string (line_length),
  status               : [STATIC] ost$status;

{ ----- }

{ This routine, P#START_REPORT_GENERATION, takes care of      }
{ initialization details. It sets the error tally to zero and }
{ prepares the report file to receive messages issued by other }
{ procedures.                                               }

PROCEDURE p#start_report_generation
  (startup_message : string ( * ));

VAR
  file_access_selection_p :
    ^ ARRAY [1 .. *] OF amt$access_selection ;

  error_count := -0 ;
```

(Continued)

(Continued)

```

    ALLOCATE file_access_selection_p : [1 .. 1] ;
    file_access_selection_p^[01].key
      := amc$open_position ;
    file_access_selection_p^[01].open_position
      := amc$open_no_positioning ;
    amc$open (report_file_name, amc$record,
      file_access_selection_p, report_file_identifier, status) ;
    FREE file_access_selection_p ;

    text_index := 1 ;
    text_line(text_index, 1) := '0' ;
    text_index := text_index + 1 ;
    p#put_m (TRUE, startup_message) ;

PROCEND p#start_report_generation ;

{ ----- }

{ Routine P#STOP_REPORT_GENERATION does wrap-up activity. The }
{ error tally is printed out at this point. }

PROCEDURE p#stop_report_generation
  (shutdown_message : string ( * ) ) ;

  VAR
    pencil : integer ,
    paper : string ( 50 ) ;

  IF error_count = 0
  THEN
    p#put_m (TRUE, 'No error has been found by the program.') ;
  ELSE
    STRINGREP (paper, pencil, 'This program has discovered ',
      error_count, ' error situation(s).') ;
    p#put_m (TRUE, paper(1, pencil)) ;
  IFEND ;
  p#put_m (TRUE, shutdown_message) ;

  amc$close (report_file_identifier, status) ;

PROCEND p#stop_report_generation ;

```



```

{ ----- }
{ P#PUT_M places the parameter message_string onto the reporting }
{ file, taking care to wrap around any long message text by }
{ splitting it into additional physical lines. Data is appended }
{ at the current character position of text_line; it doesn't }
{ automatically start in column 1. The parameter new_line_flag }
{ tells whether or not end-of-line should follow the message. }
{ Any unprintable character is translated into '?'. }

```

```

PROCEDURE p#put_m (new_line_flag : boolean ;
                  message_string : string ( * <= 321 )) ;

```

```

VAR

```

```

  garbage_eliminator_table :
    [s#dry_storage_area, STATIC, READ]
    string ( 256 ) := '?????????????????????????????????????'
    CAT ' !"#%&'()*+,-./0123456789:;<=>?@'
    CAT 'ABCDEFGHIJKLMNopqrstuvwxyz[ ]^_`'
    CAT 'abcdefghijklmnopqrstuvwxy0 }~'
    CAT '?????????????????????????????????????????????????'
    CAT '?????????????????????????????????????????????????'
    CAT '?????????????????????????????????????????????????'

```

```

  string_position_locator : 1 .. line_length,
  { Dummy variables, not used. }
  file_byte_address_x     : amt$file_byte_address,
  status_x                : ost$status;

```

```

IF (text_index + STRLENGTH(message_string) - 1) = line_length

```

```

THEN

```

```

  #TRANSLATE (garbage_eliminator_table, message_string,
    text_line(text_index, STRLENGTH(message_string)));
  text_index := text_index + STRLENGTH(message_string);
  amp$put_next (report_file_identifer, ^text_line,
    text_index - 1, file_byte_address_x, status_x);
  {Resets index }
  text_index := 1;
  {Blank fill }
  text_line(1, line_length) := ' ' ;
  {Column 1 is the carriage control character. }
  text_index := text_index + 1 ;

```

(Continued)

(Continued)

```

ELSEIF (text_index + STRLENGTH(message_string) - 1) < line_length
  THEN
    #TRANSLATE (garbage_eliminator_table, message_string,
      text_line(text_index, STRLENGTH(message_string)));
    text_index := text_index + STRLENGTH(message_string);
    IF new_line_flag
      THEN
        amp$put_next (report_file_identifier, ^text_line,
          text_index - 1, file_byte_address_x, status_x);
        text_index := 1 ;
        text_line(1, line_length) := ' ' ;
        text_index := text_index + 1 ;
      IFEND ;
ELSEIF (text_index + STRLENGTH(message_string) - 1) > line_length
  THEN
    string_position_locator := line_length - text_index + 1 ;
    #TRANSLATE (garbage_eliminator_table,
      message_string(1, string_position_locator),
      text_line(text_index, string_position_locator)) ;
    text_index := text_index + string_position_locator ;
    amp$put_next (report_file_identifier, ^text_line,
      text_index - 1, file_byte_address_x, status_x) ;
    text_index := 1 ;
    text_line(1, line_length) := ' ' ;
    text_index := text_index + 1 ;
    p#put_m (new_line_flag,
      message_string(string_position_locator + 1, *)) ;
  IFEND ;
PROCEND p#put_m ;

```

(Continued)

```
{----- }
```

```
{ This routine looks at the global status variable. If status }
{ is not normal, the global error counter is incremented and a }
{ formatted message sent to the error listing file. To prevent }
{ excessive printout, all error message reporting is suppressed }
{ when the error counter reaches 333. }
}
```

```
PROCEDURE [INLINE] p#inspect_status_variable ;
  IF NOT status.normal
  THEN
{ bump error counter }
  error_count := error_count + 1 ;
  IF error_count < 333
  THEN
    p#display_status_variable ;
  ELSEIF error_count = 333
  THEN
{ issue the message }
    p#put_m (TRUE,
  'Error_Count = 333. Further message reporting is turned off. ');
    IFEND ;
  IFEND ;
PROCEND p#inspect_status_variable ;

{----- }
```

```
{ The P#DISPLAY_STATUS_VARIABLE routine formats the status }
{ record in the global status variable using the message template, }
{ and then appends the completed diagnostic message onto the }
{ report listing file. }
}
```

```
PROCEDURE p#display_status_variable ;

VAR
  annotation : string ( 17 ) ,
  message    : ost$status_message ,
  line_count : ^ ost$status_message_line_count ,
  line_size  : ^ ost$status_message_line_size ,
  line_text  : ^ ost$status_message_line ,
  pointer    : ^ ost$status_message ,
  status_v   : ost$status ;
```

(Continued)

(Continued)

```

IF status.normal
THEN
  p#put_m (TRUE, 'NORMAL STATUS') ;
ELSE
  osp$format_message (status, osc$explain_message_level,
    line_length - 1 - STRLENGTH(annotation),
    message, status_v) ;
  IF NOT status_v.normal
  THEN
    p#put_m (TRUE, 'Unable to convert status message
      in routine P#DISPLAY_STATUS_VARIABLE.') ;
  ELSE
    annotation := ' error_status--> ' ;
    pointer := ^message ;
    RESET pointer ;
    NEXT line_count IN pointer ;
    WHILE line_count^ > 0 DO
      NEXT line_size IN pointer ;
      NEXT line_text : [line_size^] IN pointer ;
      p#put_m (FALSE, annotation) ;
      p#put_m (TRUE, line_text^) ;
      line_count^ := line_count^ - 1 ;
      annotation := '          --> ' ;
    WHILEND ;
  IFEND ;
IFEND ;

PROCEND p#display_status_variable ;

{ ----- }
?? PUSH (LIST := OFF) ??
*copyc amp$close
*copyc amp$open
*copyc amp$put_next
*copyc osp$format_message
?? POP ??

```

Index





Index

A

- Abnormal status
 - Processing for file interface calls 7-7
- Abort_line_character attribute 5-12
- Absolute pointer 8-10
- Access control entry 3-17
 - For file
 - Changing 3-22
 - Creation 3-22
 - Deletion 3-24
 - For subcatalog
 - Changing 3-25
 - Creation 3-25
 - Deletion 3-28
- Access information 7-1
 - Retrieval 7-9
- Access log 3-16
- Access modes 3-16
 - Validation for attach 3-17
 - Validation for open 7-6
- Access permissions 3-17
- Access validation during open operation 7-6
- Accessing file data 6-1
- Access_level attribute 6-16
- Access_mode attribute 6-17
- Address space 8-3
- Address translation 8-3
- ALLOCATE statement 8-1
- Allocating a heap 8-10
- Alphabetic character A-1
- Alphanumeric character A-1
- Alternate key example 10-83
- Alternate index A-1
- Alternate key A-1
- Alternate keys 10-47
 - Alternate key index 10-47
 - Concatenated key example 10-54
 - Definition 10-48
 - Duplicate key values 10-48
 - File positioning 10-69
 - Null suppression 10-51
 - Repeating groups 10-55
 - Sparse key control 10-52
- AM declarations C-1
- AMP\$ABANDON_KEY_DEFINITION procedure 10-67
- AMP\$ACCESS_METHOD procedure D-13
- AMP\$ADD_TO_FILE_DESCRIPTION procedure D-11
- AMP\$APPLY_KEY_DEFINITION procedure 10-65
- AMP\$CLOSE procedure 7-5
- AMP\$COPY_FILE procedure 11-9
- AMP\$CREATE_KEY_DEFINITION procedure 10-58
- AMP\$DELETE_KEY procedure 10-35
- AMP\$DELETE_KEY_DEFINITION procedure 10-64
- AMP\$FETCH procedure 6-15
- AMP\$FETCH_ACCESS_INFORMATION procedure 7-16
- AMP\$FETCH_FAP_POINTER procedure D-16
- AMP\$FILE procedure 6-5
- AMP\$FLUSH procedure 9-31
- AMP\$GET_DIRECT procedure 9-21
- AMP\$GET_FILE_ATTRIBUTES procedure 6-13
- AMP\$GET_KEY procedure 10-25
- AMP\$GET_KEY_DEFINITIONS procedure 10-75
- AMP\$GET_NEXT procedure 9-23
- AMP\$GET_NEXT_KEY procedure 10-28
- AMP\$GET_NEXT_PRIMARY_KEY_LIST procedure 10-81
- AMP\$GET_PARTIAL procedure 9-26
- AMP\$GET_PRIMARY_KEY_COUNT procedure 10-78

- AMP\$GET_SEGMENT_POINTER 8-6
 - AMP\$OPEN procedure 7-2
 - AMP\$PUT_DIRECT procedure 9-33
 - AMP\$PUT_KEY procedure 10-16
 - AMP\$PUT_NEXT procedure 9-35
 - AMP\$PUT_PARTIAL procedure 9-37
 - AMP\$PUTREP procedure 10-31
 - AMP\$REPLACE_KEY procedure 10-33
 - AMP\$RETURN procedure 2-7
 - AMP\$REWIND procedure 9-17
 - AMP\$SEEK_DIRECT procedure 9-11
 - AMP\$SELECT_KEY procedure 10-74
 - AMP\$SET_FILE_INSTANCE_ABNORMAL procedure D-20
 - AMP\$SET_SEGMENT_EOI procedure 8-18
 - AMP\$SET_SEGMENT_POSITION procedure 8-20
 - AMP\$SKIP procedure 9-18
 - AMP\$SKIP_TAPE_MARKS procedure 4-9
 - AMP\$START procedure 10-21
 - AMP\$STORE procedure 6-8
 - AMP\$STORE_FAP_POINTER procedure D-15
 - AMP\$VALIDATE_CALLER_PRIVILEGE procedure D-8
 - AMP\$WRITE_END_PARTITION procedure 9-39
 - AMP\$WRITE_TAPE_MARK procedure 4-12
 - ANSI fixed-length record type 9-2
 - With user-specified blocking 9-5
 - Append access 3-16
 - Append open position 7-6
 - Application_info
 - Attribute 6-17
 - Parameter 3-21
 - ASCII character set B-1
 - ASCII6_FOLDED collating sequence E-6
 - ASCII6_STRICT collating sequence E-7
 - Assigning
 - Cell pointer to data structure 8-8
 - File to device class 2-1
 - File to null device class 2-5
 - Asynchronous terminal classes 5-22
 - Attaching a file with a file reference 3-30
 - Attaching a file with PFP\$ATTACH 3-29
 - Attaching a permanent file 3-29
 - Attribute definition calls 6-4
 - Valid attributes 6-6
 - Attribute descriptions
 - File attributes 6-16
 - Terminal attributes 5-12
 - Attribute identifier 6-4
 - Attribute sources 6-10
 - Attribute specification
 - For returning values 6-9
 - For setting values 6-4
 - Audience 7
 - AV declarations C-46
 - Average_record_length attribute 6-18
- B**
- Backspace_character attribute 5-12
 - Backward skip
 - By records or partitions 9-15
 - By tapemarks 4-8
 - Beginning of information
 - Glossary definition A-1
 - Usage 9-1
 - Bit A-1
 - Block A-1
 - Block size for indexed sequential file 10-13
 - Blocking 9-4
 - Block_number for instance of open 7-10
 - Block_type attribute 6-18
 - BOI 9-1
 - Boundary conditions effect on skip operation 9-16

Burstable_form page format 6-37

Busy status 3-31

Byte A-1

Byte addressable file

Copy 11-4

Creation using sequential access calls 9-8

Example 9-12

Glossary definition A-1

Processing using random access calls 9-9

C

Call block D-6

Call bracket 6-41

Call statement 1-6

Calling a file interface procedure 1-6

Calls passed to a FAP D-2

Cancel_line_character attribute 5-12

Carriage control characters 5-31

Carriage_return_idle attribute 5-12

Catalog 3-1

Access control entry

Creation 3-25

Deletion 3-28

Creation 3-13

Deletion 3-15

Glossary definition A-2

Name A-2

Path 3-1

CDC variable record type 9-2

Cell pointer 8-8

Changing

Access control entry information

For file 3-20

For subcatalog 3-25

File attributes

New file 6-2

Old file 6-3

Open file 6-8

File entry information 3-9

Terminal attributes

After the file is open 5-8

Before the file is open 5-6

Before the file is requested 5-2

Character A-2

Character conversion 4-6

Attribute 6-19

Character set B-2

Checking the completion status 1-7

CLC\$CURRENT_COMMAND_OUTPUT file 2-8

CLC\$JOB_COMMAND_INPUT file 2-8

CLC\$JOB_COMMAND_RESPONSE file 2-8

CLC\$JOB_INPUT file 2-8

CLC\$JOB_OUTPUT file 2-8

CLC\$NULL_FILE file 2-8

Close operation A-2

Closing a file 7-1

CLP\$CREATE_FILE_

CONNECTION procedure 2-9

CLP\$DELETE_FILE_

CONNECTION procedure 2-10

COBOL6_FOLDED collating sequence E-9

COBOL6_STRICT collating sequence E-10

Code conversion

Tape files 5-6

Terminal file

Input 5-28

Output 5-30

Collated key A-2

Collate_table attribute 6-19

Collate_table_name attribute 6-20

Collation table

Creation within a FAP E-5

Glossary definition A-2

System-defined E-1

User-defined E-3

Common CYBIL procedures F-1

Completion status 1-7

Concatenated keys 10-54

Concurrent file attaches 3-29

Condition code 1-7

Condition identifier 1-7

Condition information 1-7

Connected file attributes 6-12

Connecting files 2-8

Consecutive tapemarks 4-11

- Continuous_form page format 6-37
 - Control access 3-16
 - Conventions 9
 - *COPYC directive 1-3
 - Copying
 - Files 11-1
 - Example 11-11
 - Procedure declaration decks 1-3
 - Tape files 4-11
 - Creating
 - Collation table
 - Within a FAP E-5
 - Within your program E-3
 - File access control entry 3-20
 - File connection 2-9
 - File cycle descriptor 3-3
 - File entry 3-3
 - Indexed sequential file 10-10
 - Subcatalog access control entry 3-25
 - Subcatalog entry 3-13
 - Creating and deleting alternate keys 10-57
 - Creation run
 - Glossary definition A-2
 - Current_byte_address for instance of open 7-10
 - CYBIL 7
 - Data storage mechanisms 8-1
 - Manual set 8
 - Procedure call statement 1-6
 - Cycle 3-1
 - Access 3-16
 - Busy status 3-3
 - Definition 3-3
 - Descriptor 3-3
- D**
- Data access 6-1
 - Data block 10-3
 - Glossary definition A-3
 - Padding 10-4
 - Record pointers 10-4
 - Split 10-5
 - Data_padding attribute 6-21
 - Deck
 - Creation and expansion 1-4
 - Glossary definition A-3
 - Default A-3
 - Default heap 8-1
 - Default terminal attributes 5-1
 - Defining a file cycle 3-3
 - Defining file attributes
 - New file 6-1
 - Old file 6-3
 - Open file 6-4
 - Defining primary keys for indexed sequential files 10-12
 - Defining record type and length for indexed sequential files 10-11
 - Deleting
 - File access control entry 3-24
 - File connection 2-10
 - File cycle descriptor 3-7
 - File entry 3-8
 - Records from an indexed sequential file 10-30
 - Subcatalog access control entry 3-28
 - Subcatalog entry 3-15
 - Density
 - Printing 6-42
 - Tape recording 4-1
 - Denying access to a file 3-20
 - Detaching a file 2-6
 - Device class
 - Assignment 2-1
 - Glossary definition A-3
 - Disconnecting a file 2-10
 - Display output 5-14
 - DISPLAY63_FOLDED collating sequence E-12
 - DISPLAY63_STRICT collating sequence E-13
 - DISPLAY64_FOLDED collating sequence E-15
 - DISPLAY64_STRICT collating sequence E-16
 - Dynamic expansion of task space 8-1

E

- EBCDIC collating sequence E-18
- EBCDIC6_FOLDED collating sequence E-23
- EBCDIC6_STRICT collating sequence E-24
- Echoplex attribute 5-13
- Embedded key A-3
- Embedded tapemarks 4-11
- Embedded_key attribute 6-21
- End-of-file indicator for tape files 4-1
- End-of-information 9-1
 - Establishing 9-1
 - Glossary definition A-3
- End-of-information byte address A-3
- End-of-volume indicator 4-1
- EOI string attribute 5-13
- Eoi_byte_address for instance of open 7-10
- Error exit procedure 7-7
- Error message template 1-8
- Error reporting D-19
- Error_count for instance of open 7-10
- Error_exit_name attribute 6-22
- Error_exit_procedure attribute 6-22
- Error_limit attribute 6-23
- Error_status for instance of open 7-11
- Estimated_record_count attribute 6-23
- Evaluating attach requests 3-30
- Even parity 5-16
- Examples
 - Alternate key 10-83
 - Byte addressable file 9-12
 - Concatenated keys 10-54
 - Duplicate key control for indexed sequential files 10-50
 - File access procedure (FAP) D-16
 - File copying 11-11
 - Indexed sequential file 10-39
 - Null suppression for alternate key index 10-51
 - Repeating groups for indexed sequential files 10-56
 - Segment access 8-11

- Sparse key control for indexed sequential files 10-53

- Exception condition
 - Glossary definition A-3
 - Information 1-7
- Exclusive access 3-18
- Execute access 3-16
- Execute bracket 6-41
- Execution ring A-3
- Expanding a source program 1-3
- Expiration date 3-5

F

- F record type 9-2
 - With user-specified blocking 9-5
- Family 3-1
 - Administrator 3-14,15
- FAP
 - Assignment D-4
 - Data structure D-13
 - Declaration D-5
 - Error reporting D-18
 - Example D-16
 - Glossary definition A-4
 - Loading D-4
 - Processing D-7
 - Security D-7
 - System calls D-11
 - Usage D-1
- FAP attribute 6-23
- FAP call block declarations C-39
- Fetching
 - File attributes 6-15
 - Terminal attributes 5-11
- Fetching access information after alternate key selection 10-71
- Field A-4
- File A-4
- File access identifier 7-1
- File access log 3-16
- File access procedure
 - Assignment D-4
 - Data structure D-13
 - Declaration D-5
- File attaching 3-29

INDEX

- File attribute
 - Descriptions 6-16
 - Glossary definition A-4
 - Set 6-1
 - Sources 6-10
 - Specification within a FAP D-10
 - File blocking 9-4
 - File characteristics 6-12
 - File closing 7-1
 - File connections 2-8
 - File copying 11-1
 - Program example 11-11
 - File cycle 3-1
 - Busy status 3-31
 - Definition 3-3
 - Descriptor 3-3
 - File entry 3-1
 - Changing 3-9
 - Contents 3-3
 - Definition 3-3
 - Deletion 3-8
 - File identifier 7-1
 - File information record 7-9
 - File interface procedure usage 1-1
 - Example 1-2
 - File management 3-1
 - File opening 7-1
 - File path 3-1
 - File position returned by get calls 9-20
 - File positioning 9-14
 - File positioning after alternate key selection 10-69
 - File reference A-4
 - File references 3-30
 - File sharing
 - Example 3-32
 - Open files 7-8
 - Permanent files 3-29
 - Segment access file 8-15
 - File storage 3-1
 - File subcatalogs 3-13
 - File_access_procedure attribute 6-23
 - File_contents attribute 6-24
 - File_length attribute 6-24
 - File_limit attribute 6-24
 - File_organization attribute 6-24
 - File_position for instance of open 7-11
 - File_processor attribute 6-25
 - File_structure attribute 6-25
 - Fixed-length record type 9-2
 - With user-specified blocking 9-5
 - Flushing A-4
 - Flushing data from memory 9-31
 - Folded collating sequence E-1
 - Forced_write attribute 6-26
 - Format effectors 5-30
 - Attribute 5-14
 - Insertion when copying a list file 11-8
 - To set print density 6-42
 - Forward skip
 - By records or partitions 9-15
 - By tapemarks 4-8
 - Frames 5-28
 - FREE statement 8-9
- ## G
- Get calls 9-20
 - Getting
 - Device class assignment 2-3
 - File attributes
 - After file is opened 6-15
 - Before file is opened 6-13
 - Segment pointer 8-6
 - Terminal attribute set
 - After file is opened 5-11
 - Before file is opened 5-7
 - Default values 5-4
 - Global_access_mode attribute 6-26
 - Global_file_address attribute 6-27
 - Global_file_name attribute 6-27
 - Global_file_position attribute 6-28
 - Global_share_mode attribute 6-28
- ## H
- HASP protocol terminals 5-19
 - Hazeltine 2000 terminals 5-19
 - Heap 8-1
 - Pointer 8-9

History 3

How to use file interface calls 1-1

How to use this manual 8

I

IBM 2780 terminals 5-19

IBM 3780 terminals 5-19

Idle characters 5-13

IF declarations C-46

IFC\$POST_PRINT_SPACE_1 5-31

IFC\$POST_PRINT_SPACE_2 5-31

IFC\$PRE_PRINT_HOME_CLEAR_ SCREEN 5-31

IFC\$PRE_PRINT_HOME_ CURSOR 5-31

IFC\$PRE_PRINT_NO_ POSITIONING 5-31

IFC\$PRE_PRINT_SPACE_1 5-31

IFC\$PRE_PRINT_SPACE_2 5-31

IFC\$PRE_PRINT_SPACE_3 5-31

IFC\$PRE_PRINT_START_OF_ LINE 5-31

IFP\$FETCH_TERMINAL procedure 5-11

IFP\$GET_DFLT_TERM_ ATTRIBUTES procedure 5-4

IFP\$GET_TERMINAL_ ATTRIBUTES procedure 5-7

IFP\$STORE_TERMINAL procedure 5-9

IFP\$TERMINAL procedure 5-2

Implicit release of file data 7-6

Index block 10-6

Glossary definition A-4

Levels 10-7

Padding 10-7

Record pointer 10-6

Split 10-7

Index block structure example 10-6

Index record A-4

Index records 10-6

Indexed sequential file

Access 10-24

Alternate keys 10-47

Copy

To a sequential file 11-7

To another indexed 11-5

Creation 10-10

Data padding 10-15

Duplicate key control

example 10-50

Example 10-39

File attributes 10-10

Glossary definition A-5

Index padding 10-15

Key types 10-12

Maintenance 10-37

Positioning by major key 10-21

Procedure declaration source

library 1-3

Processing 10-18

Program example 10-40

Record sorting by primary

key 10-2

Structure 10-3

Writing records 10-15

Index_levels attribute 6-29

Index_padding attribute 6-29

Initializing file space 9-9

Input blocks 5-28

Input line 5-28

Input_device attribute 5-13

Instance of open 7-1

Glossary definition A-5

Integer key A-5

Interactive conditions 5-33

Interactive terminal file

assignment 5-1

Internal file label 10-3

Internal_code attribute 6-30

J

Job A-5

Job environment attributes 6-11

Job library list A-5

Job log entry for tape assignment 4-5

K

Key

- Glossary definition A-5

- Usage 10-1

Key list A-5

- Key types for indexed sequential files 10-12

- Key_length attribute 6-30

- Key_position attribute 6-31

- Key_type attribute 6-31

L

- Label_type attribute 6-31

- Last_access_operation for instance of open 7-11

- Last_operation attribute 6-32

- Last_op_status for instance of open 7-11

- Layer number D-5

- Line feeds 5-28

- Line folding 5-32

- Line_feed_idle attribute 5-13

- Line_number attribute 6-35

- List attributes 6-42

- List file copying 11-8

- \$LOCAL file catalog 3-1

- Local file management 2-1

- Local file name 2-1

- Glossary definition A-5

- Log 3-16

- Logical file structure 9-1

- Logical lines 5-31

M

- Magnetic tape management 4-1

- Maintaining an indexed sequential file 10-37

- Major key

- Glossary definition A-6

- Manual

- Audience 7

- Mass storage

- Blocks 9-4

- Glossary definition A-6

- Master catalog 3-1

- Max_block_length attribute 6-35

- Max_record_length attribute 6-35

- Memory access 8-3

- Message template 1-8

- Message_control attribute 6-36

- Min_block_length attribute 6-36

- Min_record_length attribute 6-36

- Modify access 3-16

- Monitoring indexed sequential file growth 10-37

- Mounting tapes 4-2

- Multiple access control entries 3-18

- Multiple file attaches 3-32

- Multivolume tape files 4-2

N

NAM

- Application block number 5-26

- NAM 5-1

- Naming convention 1-9

- Network Access Method 5-1

- Network Operating System/Virtual Environment 6

- New file 2-1

- Attributes 6-1

- NEXT statement 8-13

- Nine-track tapes 4-1

- No parity 5-16

- No_format_effectors attribute 5-14

- Nonembedded key A-6

- Null device class 2-4

- \$NULL file 2-4

- Null Suppression

- Glossary definition A-6

- Null_attribute

- File attribute 6-36

- Terminal attribute 5-14

O

- Odd parity 5-16

- Old file 6-1

- Attributes 6-3

- Old files 6-3
 - Terminal attribute values 5-1
- Open operation A-6
- Opening a file 7-1
- Open_position attribute 6-37
- Optional key attribute record for indexed sequential files 10-60
- Ordering manuals 10
- Organization 8
- OS declarations C-55
- OSC\$NULL_NAME 3-2
- OSV\$ASCII6_FOLDED collating sequence E-6
- OSV\$ASCII6_STRICT collating sequence E-7
- OSV\$COBOL6_FOLDED collating sequence E-9
- OSV\$COBOL6_STRICT collating sequence E-10
- OSV\$DISPLAY63_FOLDED collating sequence E-12
- OSV\$DISPLAY63_STRICT collating sequence E-13
- OSV\$DISPLAY64_FOLDED collating sequence E-15
- OSV\$DISPLAY64_STRICT collating sequence E-16
- OSV\$EBCDIC collating sequence E-18
- OSV\$EBCDIC6_FOLDED collating sequence E-23
- OSV\$EBCDIC6_STRICT collating sequence E-24
- Output_device attribute 5-14
- Output_flow_control attribute 5-15
- Overriding
 - Device class assignment 2-2
 - File attribute values
 - New files 6-2
- P**
- PACKED data structures 8-4
- Padding
 - Data blocks 10-4
 - Fixed-length records 9-29
 - Glossary definition A-6
 - Index blocks 10-7
 - Tape file blocks 4-6
 - User-specified blocks
 - Containing F records 9-5
 - Containing U records 9-6
 - Padding_character attribute 6-37
 - Page_format attribute 6-37
 - Page_length
 - File attribute 6-38
 - Terminal attribute 5-15
 - Usage 5-32
 - Page_wait
 - Terminal attribute 5-15
 - Usage 5-32
 - Page_width
 - File attribute 6-38
 - Terminal attribute 5-15
 - Usage 5-32
 - Paper tape
 - Input 5-13
 - Output 5-14
 - Parameter list 1-6
 - Parameter types 1-10
 - Parity attribute 5-16
 - Partial record read 9-26
 - Partition A-6
 - Partition delimiter 9-1
 - Writing 9-38
 - Path 3-1
 - Glossary definition A-6
 - Specification 3-2
 - Pause break condition 5-33
 - Pause_break_character attribute 5-16
 - Permanent file
 - Attaching 3-29
 - Creation 3-3
 - Deletion 3-8
 - Permanent_file attribute 6-39
 - Permit selections set 3-17
 - Permitting access to a file 3-20
 - PF declarations C-57
 - PFC\$FAMILY_NAME_INDEX 3-2
 - PFC\$MASTER_CATALOG_NAME_INDEX 3-2

- PFC\$SUBCATALOG_NAME_INDEX 3-2
 - PF\$ATTACH procedure 3-34
 - PF\$CHANGE procedure 3-10
 - PF\$DEFINE procedure 3-4
 - PF\$DEFINE_CATALOG procedure 3-14
 - PF\$DELETE_CATALOG_PERMIT procedure 3-28
 - PF\$DELETE_PERMIT procedure 3-24
 - PF\$PERMIT procedure 3-20
 - PF\$PERMIT_CATALOG procedure 3-25
 - PF\$PURGE procedure 3-7
 - PF\$PURGE_CATALOG procedure 3-15
 - PM declarations C-62
 - Pointer
 - Glossary definition A-7
 - Positioning 10-19
 - Indexed sequential file by major key 10-20
 - Mass storage files 9-14
 - By records or partitions 9-14
 - Tape files 4-7
 - Preface 7
 - Preserved attributes 6-1
 - Previous_record_address for instance of open 7-13
 - Previous_record_length for instance of open 7-13
 - Primary key 10-1
 - Defining 10-12
 - Glossary definition A-7
 - Print density attribute 6-42
 - Printer output 5-14
 - Procedure call statement 1-6
 - Procedure declaration decks 1-3
 - Procedure parameter list 1-6
 - Process identifier 1-8
 - Process virtual address 8-3
 - Processing for file interface calls 7-7
 - Returned in status 1-8
 - Processor names 6-25
 - Program examples
 - Byte addressable file 9-12
 - File copying 11-11
 - Indexed sequential file 10-40
 - Segment access 8-11
 - Program interface 1-1
 - Program library list A-7
 - Programs for use in checking
 - procedure completion status F-1
 - Prompt_file attribute 5-16
 - Prompt_file_id attribute 5-17
 - Prompt_string attribute 5-18
 - Purging
 - File 3-8
 - File cycle 3-7
 - Subcatalog 3-15
 - PUSH statement 8-1
 - PVA 8-3
- Q**
- Queuing terminal input 5-29
- R**
- Random access
 - Glossary definition A-7
 - Usage 9-9
 - Read access 3-16
 - Read bracket 6-41
 - Reading
 - File shared by other tasks 7-8
 - Packed data structures in a segment access file 8-4
 - Records
 - From a byte addressable file 9-20
 - From a sequential file 9-20
 - From an indexed sequential file 10-24
 - Reading records after alternate key selection 10-69
 - Real memory 8-3
 - Record
 - Glossary definition A-7

- Record access
 - For byte addressable file
 - organization 9-1
 - For indexed sequential file
 - organization 10-1
 - For sequential file
 - organization 9-1
 - Record types 9-2
 - Recording density 4-1
 - Record_limit attribute 6-39
 - Records 9-1
 - Records_per_block attribute 6-40
 - Record_type attribute 6-39
 - Recreating an indexed sequential file 10-37
 - Related manuals 2
 - Relative pointers 8-10
 - Releasing
 - File data upon open 7-6
 - Heap space 8-9
 - Temporary file space 2-6
 - Replacing records in an indexed sequential file 10-30
 - Repositioning a file 9-18
 - Requesting
 - Null file 2-5
 - Tape file 4-3
 - Terminal file 5-5
 - RESET statement
 - For heap 8-9
 - For sequence 8-13
 - Residual_skip_count for instance of open 7-13
 - Resource limit condition
 - processing 5-33
 - Retrieving
 - Access information 7-9
 - Connected file attributes 6-12
 - Device class assignment 2-3
 - File attributes 6-9
 - File characteristics 6-12
 - Terminal attribute set
 - After file is opened 5-11
 - Before file is opened 5-7
 - Default values 5-4
 - Retrieving alternate index information 10-73
 - Returned attributes 6-11
 - Returning a file 2-6
 - Return_option attribute 6-40
 - Revision record 3
 - Rewind A-7
 - Rewinding files 9-17
 - Ring A-7
 - Ring attribute A-7
 - Ring number validation 7-6
 - Ring_attributes attribute 6-41
 - RM declarations C-62
 - RMP\$GET_DEVICE_CLASS
 - procedure 2-3
 - RMP\$REQUEST_NULL_DEVICE
 - procedure 2-5
 - RMP\$REQUEST_TAPE
 - procedure 4-3
 - RMP\$REQUEST_TERMINAL
 - procedure 5-6
 - Run-time stack 8-1
 - RUN_CHECKS CYBIL
 - parameter 8-9
- S**
- SCL procedure A-8
 - SCU deck creation and expansion 1-3
 - Segment A-8
 - Segment access 8-1
 - Program example 8-11
 - Segment attributes 8-4
 - Segment length 8-4
 - Segment pointer 8-5
 - Sequence pointer 8-13
 - Sequential access
 - Glossary definition A-8
 - Usage 9-7
 - Sequential file copy
 - To an indexed sequential file 11-3
 - To another sequential file 11-2
 - Sequential file organization A-8
 - Setting
 - Current byte address
 - For byte addressable file 9-11
 - For segment 8-17

- End-of-information address for segment
 - Using 8-15,17
 - File attribute values
 - New file 6-1
 - Old file 6-3
 - Open file 6-4
 - Within a FAP D-10
 - Terminal attribute values 5-1
 - Share requirements set 3-18
 - Share selections 3-18
 - Sharing
 - File with write access 3-30
 - Open files 7-8
 - Permanent files 3-32
 - Segment access file 8-15
 - Shorten access 3-16
 - Skipping
 - By partitions 9-15
 - By records 9-15
 - By tapemarks 4-7
 - Source library files 1-4
 - Source text preparation example 1-4
 - Sparse key control for alternate keys 10-52
 - Special_editing attribute 5-19
 - Statement_identifier attribute 6-41
 - Status variable 1-7
 - Storing
 - File attributes 6-8
 - Terminal attributes 5-9
 - Strict collating sequence E-1
 - Structural attributes 6-1
 - Subcatalog 3-1
 - Definition 3-13
 - Deletion 3-15
 - Subject file 2-8
 - Submitting comments 10
 - Synchronous terminal classes 5-24
 - SYSTEM.COMMON.
 - PSF\$EXTERNAL_INTERFACE_SOURCE 1-3
 - SYSTEM.CYBIL.OSF\$PROGRAM_INTERFACE 1-3
 - System-defined collation tables E-1
 - System file connections 2-8
 - System naming convention 1-9
 - System services 1-1
 - System-specified blocking 9-4
- ## T
- Tape file
 - Attributes 4-6
 - Blocks 9-4
 - Character conversion 4-6
 - Padding 4-6
 - Positioning 4-7
 - Requests 4-1
 - Tape management 4-1
 - Tapemark
 - Glossary definition A-8
 - Usage 4-1
 - Target file 2-8
 - Task A-8
 - Task's address space 8-3
 - Tektronix 4010 terminals 5-19
 - Teletypewriters 5-19
 - Temporary attributes 6-1
 - Temporary file storage 3-1
 - Terminal attribute
 - Default values 5-22
 - Descriptions 5-12
 - Glossary definition A-8
 - Set 5-1
 - Sources 5-5
 - Terminal class A-8
 - Terminal conditions 5-33
 - Terminal file
 - Access information 5-26
 - Attributes 5-26
 - Processing considerations 5-26
 - Requests 5-5
 - Terminal input 5-28
 - Terminal management 5-1
 - Terminal output 5-30
 - Terminal_class attribute 5-19
 - Terminal_name attribute 5-20
 - Terminate break condition 5-33
 - Terminate_break_character attribute 5-20
 - Title insertion frequency 6-38
 - Transparent_delim_selection attribute 5-20

- Transparent_end_character
 - attribute 5-20
- Transparent_end_count
 - attribute 5-20
- Transparent_mode attribute 5-20
- Trivial error limit 6-23
- Type checking 1-6
- Typed ahead input 5-29

U

- U record type 9-3
 - With system-specified blocking 9-4
 - With user-specified blocking 9-6
- Uncollated key A-9
- Undefined record type 9-3
 - With system-specified blocking 9-4
 - With user-specified blocking 9-6
- Unlabeled tape files 4-1
- Update run 10-16
- Updating an alternate index 10-70
- Usage selections 3-17
- User groups 3-17
- User-specified blocking 9-4
- User_info attribute 6-42
- Using alternate keys 10-68
- Using alternate keys to access
 - indexed sequential files 10-47
- Using file interface procedures 1-1
- Using indexed sequential files 10-1

V

- V record type 9-2
 - With user-specified blocking 9-5
- Valid attributes for each file attribute
 - call 6-7
- Validating caller privilege within a
 - FAP D-8
- Variable record type 9-2
 - With user-specified blocking 9-5
- Verifying preserved attribute
 - values 6-3
- Vertical_print_density attribute 6-42
- Virtual address translation 8-3
- Virtual memory access 8-3

- Volume switching 4-2
- Volume_number for instance of
 - open 7-14
- Volume_position for instance of
 - open 7-14

W

- Wait option 3-33
- Working storage area
 - Glossary definition A-9
 - Usage 9-1
- Write access 3-16
- Write bracket 6-41
- Write ring 4-1
- Writing
 - File shared by tasks 7-9
 - Partition delimiters 9-38
 - Records 9-29
 - Longer than the working
 - storage area 9-30
 - Tapemark 4-12
- Writing a byte addressable file using
 - sequential access calls 9-8
- Writing a record to an indexed
 - sequential file 10-15

Z

- Zero parity 5-16



CYBIL for NOS/VE, File Interface 60464114 B

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

Which Do You Also Have?

- Any SCL Manuals
- CYBIL System Interface
- CYBIL Language Definition

What programming languages do you use? _____

Which are helpful to you? Procedures Index (inside covers) Glossary Related Manuals page

Character Set Other: _____

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Is the manual easy to read (print size, page layout, and so on)?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Is it easy to understand?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Is the order of topics logical?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Are there enough examples?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Are the examples helpful? (<input type="checkbox"/> Too simple <input type="checkbox"/> Too complex)
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Is the technical information accurate?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Can you easily find what you want?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Do the illustrations help you?
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? Yes No Continue on other side

From:

Name _____ Company _____

Address _____ Date _____

_____ Phone No. _____

Please send program listing and output if applicable to your comment.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
ARH219
4201 North Lexington Avenue
Saint Paul, Minnesota 55112



FOLD
Comments (continued from other side)

FOLD