CYBIL for NOS/VE System Interface

CONTROL DATA

Usage

60464115

CYBIL for NOS/VE System 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 60464115

Manual History

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

Revision B removed the Procedure Declaration subsections of each CYBIL procedure description. Each parameter's CYBIL type was taken from the Procedure Declaration subsection and incorporated into the parameter's description. Chapter 1 was rewritten to improve usability. The SLC commands GENMT and GENPDT were removed; also any access method procedure commands (AMP) were removed, as the AM product is documented in the CYBIL File Interface manual.

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	October 1983

Contents

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 set of CYBIL procedures that serve as a program interface between CYBIL programs and the operating system. These CYBIL procedures are presented in two manuals: the CYBIL File Interface manual, and this, the CYBIL System Interface manual.

Audience

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

To use the procedure calls described in this manual, you must copy decks from a system source 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 you are familiar with the System Command Language (SCL). You can perform many system functions described in this manual using either SCL commands or CYBIL procedure calls. All 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 and SCL Language Definition manuals.

Other manuals that relate to this manual are shown on the Related Manuals diagram on the reverse side of the title page.

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.
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.
blue	Within interactive terminal examples, user input is shown in blue.
`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.

How to Use System Interface Calls 1

How to Use System Interface Calls 1

NOS/VE provides a set of CYBIL procedures by which programs can request system services. System services are functions which supply information to application programs. These services are supported by the operating system.

This manual describes the system interface portion of the NOS/VE-supplied CYBIL procedures. It provides the CYBIL programmer with the information required to make calls to system interface procedures in CYBIL programs.

Using System Interface Procedures

Each CYBIL system interface procedure resides as an externally referenced (XREF) procedure declaration in a deck on a system source library. In general, to use a system 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 system interface procedure. System-defined names are shown in uppercase letters; user-defined names in lowercase letters.

Copying Procedure Declaration Decks

To use a system interface procedure in a CYBIL module, the module must include an SCU *COPYC directive to copy the XREF procedure from a system source library. The XREF procedure declarations for all system interface calls are stored in decks on the source library file $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE.

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

As shown in figure 1-1, the *COPYC directive begins in column 1, specifies the name of the procedure to be copied, and follows the module statement. In your CYBIL module, you will need only one *COPYC directive for each unique call to a system interface procedure. For example, if the module in figure 1-1 had called the PMP$GET_TIME procedure more than once, one *COPYC directive to copy the XREF PMP$GET_TIME procedure deck would suffice.

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

Procedure declaration decks list the required parameters as well as the valid parameter types that must be listed on a call to a system interface procedure. When a CYBIL program is being compiled, the parameters specified on a call to a system 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 system interface procedure do not match the parameters and parameter types defined in the XREF procedure declaration, the program compilation will be unsuccessful. After the module in figure 1-1 is compiled, the XREF procedure declaration will be included in the source listing.

An example of an XREF procedure declaration is shown later in this chapter under the subheading, Calling a System Interface Procedure.

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

For 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 in the user's master catalog at the end of the SCU session.

3. Creates a deck on the source library named MY_PROGRAM. The deck MY_PROGRAM now contains the CYBIL source program which was contained in the local file, SOURCE_FILE.

4. Expands the MY_PROGRAM deck. Decks specified on *COPYC directives are copied from the alternate base library file, $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE. 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. (If WRITE_LIBRARY=FALSE is used to end the SCU session, no result library file is written; however, the expanded source text remains available on the COMPILE file).

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.

For example, the procedure declaration for the PMP$GET_TIME procedure is as follows:

```
PROCEDURE [XREF] pmp$get_time
   (format: ost$time_formats;
  VAR time: ost$time;
  VAR status: ost$status);
```

This declaration indicates that a call to the procedure must specify three parameters in its parameter list. The first parameter must specify an input value of type OST$TIME_FORMATS; the second parameter must specify a variable of type OST$TIME; and the third parameter must specify a variable of type OST$STATUS.

The required parameter types for each parameter on a system interface procedure are listed with the parameter name in each procedure's individual description format. All parameter types are also listed alphabetically in Appendix C.

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

Exception Condition Information

When the procedure completes abnormally, NOS/VE returns additional information about the exception condition that occurred. The following fields of the status 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

Error code that uniquely identifies the error (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 a message is formatted using this status record.

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 an OSP$GET_STATUS_SEVERITY procedure call.

Table 1-1. Product Identifiers for System Interface Calls

Product Identifier	Product Function
AV	Accounting and validation
CL	Command language
IC	Interstate communication
IF	Interactive file and terminal management
JM	Job management
MM	Memory management
OF	Operator facility
OS	Operating system
PF	Permananet file management
PM	Program management

Program Services 2

Program Services 2

The program services described in this chapter provide the means to retrieve information maintained by the operating system; change job sense switch settings; and send messages to the job log, the system operator, or the job status display.

Date and Time Retrieval

NOS/VE uses two date and time formats: legible and compact. Legible format is used to display the date and time; compact format is used to compute a new date and time.

The following procedures return the current date and time.

PMP$GET_DATE

Returns the current date in a legible format.

PMP$GET_TIME

Returns the current time in a legible format.

PMP$GET_LEGIBLE_DATE_TIME

Returns the current date and time in a legible format.

PMP$GET_COMPACT_DATE_TIME

Returns the current date and time in a compact format.

The PMP$COMPUTE_DATE_TIME procedure computes a new compact date and time from a base date and time in compact format and increments the value for each date and time field.

The following procedures change the compact date or time format to a legible date or time format.

PMP$FORMAT_COMPACT_DATE

Reformats a date from a compact format to a legible format.

PMP$FORMAT_COMPACT_TIME

Reformats a time from a compact format to a legible format.

PMP$GET_TIME

Purpose Returns the current time in legible format.

Format **PMP$GET_TIME (format, time, status)**

Parameters **format**: ost$time_formats;

Format in which time is returned.

OSC$AMPM_TIME

Format hour:minute AM or PM.
For example, 1:15 PM.

OSC$HMS_TIME

Format hour:minute:second.
For example, 13:15:21.

OSC$MILLISECOND_TIME

Format hour:minute:second:millisecond.
For example, 13:15:21:453.

OSC$DEFAULT_TIME

Default format selected during system installation.

time: VAR of ost$time;

Time returned.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifier pme$invalid_time_format

time_format: ost$time_formats;

Format in which time is returned.

OSC$AMPM_TIME

Format hour:minute AM or PM.
For example, 01:15 PM.

OSC$HMS_TIME

Format hour:minute:second.
For example, 13:15:21.

OSC$MILLISECOND_TIME

Format hour:minute:second:millisecond.
For example, 13:15:21:453.

OSC$DEFAULT_TIME

Default format selected during system installation.

time: VAR of ost$time;

Time returned.

status: VAR of ost$status;

Status record.

Condition Identifiers

pme$invalid_date_format
pme$invalid_time_format

PMP$COMPUTE_DATE_TIME

Purpose Computes a new compact date and time from a base date and time also in compact format; increments value for each field.

Format **PMP$COMPUTE_DATE_TIME (base, increment, result, status)**

Parameters **base**: ost$date_time;

Base date and time returned by the PMP$GET_COMPACT_DATE_TIME procedure.

increment: pmt$time_increment;

Increment values.

Field	Content
year	Increment value for year (integer).
month	Increment value for month (integer).
day	Increment value for day (integer).
hour	Increment value for hour (integer).
minute	Increment value for minute (integer).
second	Increment value for second (integer).
millisecond	Increment value for millisecond (integer).

result: VAR of ost$date_time;

New date and time in compact format.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers pme$compute_overflow
pme$invalid_year

Remarks The increment values can be any combination of positive and negative integers.

PMP$FORMAT_COMPACT_TIME

Purpose Reformats a time from compact format to legible format.

Format **PMP$FORMAT_COMPACT_TIME (date_time, format, time, status)**

Parameters **date_time**: ost$date_time;

Date and time returned by the PMP$GET_COMPACT_DATE_TIME procedure.

format: ost$time_ formats;

Legible time format.

OSC$AMPM_TIME

Format hour:minute AM or PM.
For example, 01:15 PM.

OSC$HMS_TIME

Format hour:minute:second.
For example, 13:15:21.

OSC$MILLISECOND_TIME

Format hour:minute:second:millisecond.
For example, 13:15:21:453.

OSC$DEFAULT_TIME

Default format selected during system installation.

time: VAR of ost$time;

Time in legible format.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$invalid_hour
pme$invalid_millisecond
pme$invalid_minute
pme$invalid_second
pme$invalid_time_format

```
{ INITIALIZE ALL VARIABLES FOR INCREMENT }
    increment.year :=0;
    increment.month := 1;
    increment.day :=0;
    increment.hour :=0;
    increment.minute :=0;
    increment.second :=0;
    increment.millisecond :=0;
    pmp$compute_date_time (base_date_time, increment,
      new_date_time, status);
    IF NOT status.normal THEN
      EXIT /time_example_block/;
    IFEND;

    pmp$format_compact_date (new_date_time,
      osc$month_date, print_date, status);
    IF NOT status.normal THEN
      EXIT /time_example_block/;
    IFEND;

    pmp$format_compact_time (new_date_time,
      osc$ampm_time, print_time, status);
    IF NOT status.normal THEN
      EXIT /time_example_block/;
    IFEND;
  END /time_example_block/

PROCEND month_ahead;
MODEND date_time_example;
```

PMP$GET_MICROSECOND_CLOCK

Purpose	Returns a 64-bit integer value.
Format	**PMP$GET_MICROSECOND_CLOCK (microsecond_ clock, status)**
Parameters	**microsecond_clock**: VAR of integer; Integer value returned. **status**: VAR of ost$status; Status record.
Condition Identifier	None.
Remarks	The value returned is the current value of the microsecond clock. Successive calls to the procedure always return different values.

PMP$GET_OS_VERSION

Purpose Returns the operating system name and version number.

Format **PMP$GET_OS_VERSION (version, status)**

Parameters **version**: VAR of pmt$os_name;

Operating system name and version number. The 22-character string returned has the following format.

NOS/VE Rnn xxxxxxxxxxx

nn

Number indicating the operating system release level.

xxxxxxxxxxx

String defined during system installation.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Job Information Retrieval

The following procedure calls are used by a task to get information about itself or about the job to which it belongs. A task is the execution of a program within a job.

Two of the calls, PMP$GET_TASK_CP_TIME and PMP$GET_TASK_ID, are dependent on the task that issues the call.

PMP$GET_ACCOUNT_PROJECT

Returns the job account and project names.

PMP$GET_JOB_MODE

Returns the job execution mode (batch or interactive).

PMP$GET_SRUS

Returns the system resource units used by job.

PMP$GET_TASK_CP_TIME

Returns the amount of central processor time currently used by the task.

PMP$GET_TASK_ID

Returns the identifier of the task within the job.

PMP$GET_USER_IDENTIFICATION

Returns the job user and family names.

PMP$GET_JOB_MODE

Purpose Returns the current execution mode of the job to which the task belongs.

Format **PMP$GET_JOB_MODE (mode, status)**

Parameters **mode**: VAR of jmt$job_mode;

Job mode.

JMC$BATCH

Batch job.

JMC$INTERACTIVE_CONNECTED

Interactive job connected to terminal input.

JMC$INTERACTIVE_CMND_DISCONNECT

Interactive job disconnected from terminal by user request (see DETACH_JOB command in SCL manual set).

JMC$INTERACTIVE_LINE_DISCONNECT

Interactive job disconnected from terminal by communications equipment failure.

JMC$INTERACTIVE_SYS_DISCONNECT

Interactive job disconnected from terminal by (recovered) system failure.

status: VAR of ost$status;

Status record.

Condition Identifier None.

PMP$GET_SRUS

Purpose Returns the current number of system resource units (SRUs) accrued by the job to which the task belongs.

Format **PMP$GET_SRUS (srus, status)**

Parameters **srus**: VAR of jmt$sru_count;

Number of SRUs (0 through JMC$SRU_COUNT_MAX).

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks Currently, the SRU value is the number of microseconds of CP time accumulated for the job in both monitor and job modes.

PMP$GET_TASK_ID

Purpose Returns the system-assigned identifier of the task.

Format **PMP$GET_TASK_ID (task_id, status)**

Parameters **task_id**: VAR of pmt$task_id;
Task identifier (0 through PMC$MAX TASK ID).

status: VAR of ost$status;
Status record.

Condition Identifier None.

Sense Switch Management

NOS/VE maintains eight local sense switch values for each job. Each switch is either set (on) or cleared (off). Initially, all sense switches for a job are cleared (off).

The PMP$MANAGE_SENSE_SWITCHES procedure can set, clear, or return the values of the job sense switches. The sense switch settings are returned as a set of integers, 1 through 8. If an integer is included in the set, its corresponding sense switch is set.

The procedure call specifies a set of switches to be set and a set of switches to be cleared. It returns the set of switches that are set at completion of the procedure.

NOTE

Do not set and clear a sense switch with the same procedure call. If a call specifies that a sense switch is to be both set and cleared, the resulting switch state is undefined at completion of the procedure.

You can determine the sense switch settings without changing them by specifying no sense switch changes on the call; the procedure returns the current sense switch settings.

Sense Switch Example

The following is the source text for a procedure declaration. The procedure returns a boolean value indicating whether the sense switch specified by the integer passed to the procedure is currently set.

```
MODULE sense_switch_example;

*copyc pmp$manage_sense_switches

    PROCEDURE sensor (switch: integer;
      VAR switch_set : boolean;
      VAR status: ost$status);

      VAR
        on, off, current : pmt$sense_switches;

      on := $pmt$sense_switches[];
      off := $pmt$sense_switches[];

      pmp$manage_sense_switches (on, off, current,
        status);
      IF NOT status.NORMAL THEN
        RETURN;
      IFEND;

      switch_set := switch IN current;

    PROCEND sensor;
MODEND sense_switch_example;
```

PMP$LOG

Purpose Enters a message in the job log.

Format **PMP$LOG (text, status)**

Parameters **text**: pmt$log_msg_text;

Text to be entered in the job log (adaptable string).

status: VAR of ost$status;

Status record. The process identifier returned is PMC$EXTERNAL_LOG_MANAGEMENT_ID.

Condition Identifiers
pme$logging_not_yet_active
pme$job_log_no_longer_active

Remarks Each entry in the job log is a record of type PMT$JOB_LOG_ENTRY. The record has the following fields.

time

Time of the log entry (type OST$MILLISECOND_TIME).

delimiter_1

Delimiter character.

origin

Process identifier indicating the source of the entry (two-character string).

delimiter_2

Delimiter character.

text

Message text as specified on the PMP$LOG call (type PMT$LOG_MSG_TEXT).

OFP$DISPLAY_STATUS_MESSAGE

Purpose Sends a job status message.

Format **OFP$DISPLAY_STATUS_MESSAGE (text, status)**

Parameters **text**: string (*);

Job status message.

status: VAR of ost$status;

Status record. The process identifier returned is OFC$OPERATOR_FACILITY_ID.

Condition Identifier ofe$message_too_long

Remarks

- The message sent by the call can appear on the operator's job display or at an interactive terminal. The message appears when the user or operator enters a DISPLAY_JOB_STATUS command.
- If the message is longer than OFC$MAX_DISPLAY_MESSAGE characters, the message is truncated to that length before it is sent. The exception condition OFE$MESSAGE_TOO_LONG is returned to the caller.

OFP$RECEIVE_FROM_OPERATOR

Purpose Receives a message the operator has sent to the task.

Format **OFP$RECEIVE_FROM_OPERATOR (wait, text, operator_id, status)**

Parameters **wait**: ost$wait;

Indicates whether the task should wait for a message or continue processing.

OSC$WAIT

Suspend execution until a message is received.

OSC$NOWAIT

Continue execution if no message is waiting.

text: VAR of ost$string;

Message.

Field	Content
size	Message length in characters (0 through OSC$MAX_STRING_SIZE, 256).
value	Message text (String of length OSC$MAX_STRING_SIZE, 256).

operator_id: VAR of oft$operator_id;

Operator identifier. Currently, the only valid operator identifier is SYSTEM_OPERATOR.

status: VAR of ost$status;

Status record. The process identifier returned is OFC$OPERATOR_FACILITY_ID.

Program Execution 3

Program Execution 3

A NOS/VE program is a set of object code modules. Program execution is the process of combining and executing the modules that compose a program.

A task is an instance of program execution. More than one task can be executing the same program at the same time. If you specify that the program be loaded from an object library rather than an object file, all tasks executing the program can share the same physical copy.

Each task has a separate virtual address space. The segment numbers assigned to the task are meaningful only for that task and are discarded after the task completes.

A task can initiate other synchronous or asynchronous tasks. It initiates a task by calling the PMP$EXECUTE procedure. It terminates a task it has initiated by calling the PMP$TERMINATE procedure. It can also suspend itself until an initiated task terminates by calling the PMP$AWAIT_TASK_TERMINATION procedure.

When an initiated task completes, its status record is returned to the calling task. The initiated task can terminate itself by returning from its starting procedure or by calling the PMP$EXIT or PMP$ABORT procedures.

Program Description

To initiate another task, a task must initialize a program description variable. The content of the program description variable is described in table 3-1. A program description lists all modules that comprise a program; it includes an object file list, a module list, an object library list, and the starting procedure for the program.

Table 3-1. Program Attributes Record (PMT$PROGRAM_ATTRIBUTES) *(Continued)*

Field	Content
number_of_ object_files	Number of files in the object file list (type PMT$NUMBER_ OF_OBJECT_FILES, 0 through PMC$MAX_OBJECT_ FILE_LIST).
number_of_ modules	Number of modules in module list (type PMT$NUMBER_ OF_MODULES, 0 through PMC$MAX_MODULE_LIST).
number_of_ libraries	Number of libraries in the object library list (type PMT$NUMBER_OF_LIBRARIES, 0 through PMC$MAX_ LIBRARY_LIST).
load_map_ file	Name of load map file (type AMT$LOCAL_FILE_NAME).
load_map_ options	Set of load map options (type PMT$LOAD_MAP_ OPTIONS, set of the following constant identifiers). PMC$NO_LOAD_MAP No load map. PMC$SEGMENT_MAP Segment map. PMC$BLOCK_MAP Block map. PMC$ENTRY_POINT_MAP Entry point map. PMC$ENTRY_POINT_XREF Entry point and external reference map.
termination_ error_level	Error severity that causes task termination (type PMT$TERMINATION_ERROR_LEVEL). PMC$WARNING_LOAD_ERRORS Terminate the load when an error of warning severity occurs. PMC$ERROR_LOAD_ERRORS Terminate the load when an error of error severity occurs. PMC$FATAL_LOAD ERRORS Terminate the load when an error of fatal severity occurs.

(Continued)

The starting procedure of a program is the name of the procedure where execution of the program begins. For a CYBIL program, the procedure name must be externally declared (have the XDCL attribute) or be declared within a PROGRAM statement. If the starting procedure is not explicitly specified, the system uses the last transfer symbol encountered during program loading as the starting procedure. A transfer symbol is generated by either a CYBIL or FORTRAN PROGRAM statement or by a COBOL PROGRAM-ID statement.

An object file list is the list of object files whose modules are to be included in the program. All modules on each of the files are included.

The program library list is the set of object libraries from which modules can be loaded for the program. It has the following components.

1. Object libraries listed in the program description. The libraries are searched in the order listed.

2. Object libraries quoted by the compiler or assembler in the object text output; the libraries are searched in the order encountered during loading. NOS/VE adds the libraries to the list before satisfying the external references of the module that quoted the libraries.

3. Job library list. (You can change the contents of the job library with the SCL command SET_PROGRAM_ATTRIBUTES.)

4. NOS/VE task services library. If desired, the task services library can be searched earlier in the search order by specifying it in the program library list in the program description. Although the task services library is actually a system table, you can reference it in the program library list using the reserved name OSF$TASK_SERVICES_LIBRARY.

The module list in a program description is a list of modules to be loaded from files in the program library list. In general, you specify a module in the module list when a required entry point name is used in more than one module in the program library list. By explicitly specifying the module, you ensure that the correct entry point is loaded.

NOTE

When specifying program names for the module_list parameter, it is important to remember to specify the program name using uppercase letters. Because CYBIL converts all names to uppercase, the NOS/VE loader will be unable to locate a program name specified in any other manner.

PMP$GET_PROGRAM_SIZE

Purpose Returns the sizes of the object file list, the module list, and the library list within the program description of the requesting task.

Format **PMP$GET_PROGRAM_SIZE (number_of_object_files, number_of_modules, number_of_libraries, status)**

Parameters **number_of_object_files**: VAR of pmt$number_of_object_files;

Number of object files in the program description (0 through PMC$MAX_OBJECT_FILE_LIST).

number_of_modules: VAR of pmt$number_of_modules;

Number of modules in the program description (0 through PMC$MAX_MODULE_LIST).

number_of_libraries: VAR of pmt$number_of_libraries;

Number of libraries in the program description (0 through PMC$MAX_LIBRARY_LIST).

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks The list sizes returned can be used to allocate a program_description variable for a PMP$GET_PROGRAM_DESCRIPTION call.

Task Parameters

When a task is initiated, two parameters are passed to the starting procedure of the task: a parameter list and a status record.

The parameter list provides input information to the task. The parameter list must be an adaptable sequence. The content of the variable depends on the requirements of the task. If the task requires no input, the parameter list is empty.

For example, a CYBIL starting procedure would have the following format.

```
PROGRAM prog_name (param_list: clt$parameter_list;
  VAR status: ost$status);
```

The status record passes the status of the completed task back to the initiating task if the task completes by returning from its starting procedure. If the task terminates by calling PMP$EXIT or PMP$ABORT, the status record specified on the call is returned to the initiating task.

PMP$EXECUTE

Purpose Initiates a task.

Format **PMP$EXECUTE (program_description, parameters, wait, task_id, task_status, status)**

Parameters **program_description**: pmt$program_description;

Program description. The parameter is an adaptable sequence that must contain a program_attributes variable; the other variables are optional. The variables are:

program_attributes

Program attributes including the presence and size of the other variables (see table 3-1).

object_file_list

List of object files (PMT$OBJECT_FILE_LIST, adaptable array of AMT$LOCAL_FILE_NAME).

module_list

List of modules (PMT$MODULE_LIST, adaptable array of PMT$PROGRAM_NAME).

object_library_list

List of object libraries (PMT$OBJECT_LIBRARY_LIST, adaptable array of AMT$LOCAL_FILE_NAME).

parameters: pmt$program_parameters;

Parameter list passed to the task (adaptable sequence).

wait: ost$wait;

Indicates whether or not the requesting task should wait until the initiated task completes.

OSC$WAIT

Suspend execution until the initiated task terminates (synchronous execution).

OSC$NOWAIT

Continue execution without waiting for the initiated task to terminate (asynchronous execution).

PMP$LOAD

Purpose Returns the address of the specified externally declared procedure within the requesting task.

Format **PMP$LOAD (name, kind, address, status)**

Parameters **name**: pmt$program_name;

Procedure or variable name externally declared in the program.

NOTE

If you are loading a CYBIL program, you must specify the name using uppercase letters. The loader does not convert lowercase letters to uppercase letters; therefore, if you specify the name using lowercase letters, the loader cannot find the name in the program library list.

---

kind: pmt$loaded_address_kind;

Address type returned.

PMC$PROCEDURE_ADDRESS

Procedure address.

PMC$DATA_ADDRESS

Data address.

address: VAR of pmt$loaded_address;

Address type and value.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

lle$entry_point_not_found
lle$insufficient_memory_to_load
lle$loader_malfunctioned
lle$premature_load_termination
lle$term_error_level_exceeded

Remarks If the procedure is not yet defined in the requesting task, it is loaded dynamically from the program library list. The address assigned to it is returned.

PMP$AWAIT_TASK_TERMINATION

Purpose Suspends the task until a task it initiated terminates.

Format **PMP$AWAIT_TASK_TERMINATION (task_id, status)**

Parameters **task_id**: pmt$task_id;

Task identifier returned by the PMP$EXECUTE call that initiated the task.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifier pme$invalid_task_id

Task Termination

When a task terminates, it returns a status record to the task_status variable specified on the PMP$EXECUTE call that initiated the task. The content of the status record depends on how the task terminated.

When the task is initiated, its status record is initialized for normal completion. If the task does not change the contents of the status record and terminates by returning from its starting procedure, the normal status record is returned to the initiating task.

The task can specify the status record it returns by either changing the contents of the status record returned when its starting procedure terminates, or by specifying a status record on a PMP$EXIT or PMP$ABORT procedure call.

The PMP$EXIT procedure terminates the task just as if the starting procedure had returned to caller, except that the task specifies the status record returned. The record can indicate either normal or abnormal status. The task should call PMP$EXIT when it cannot perform its function due to an error in the job environment or in the parameter list passed to it. The condition code returned in the status record should notify the calling task of the error.

Like the PMP$EXIT procedure, the PMP$ABORT procedure returns the status record specified on its call. It should be used when the task detects an internal failure. The PMP$ABORT procedure calls the debugging program to execute the contents of the abort file before returning to the calling task. To use this feature, you must specify the name of the abort file in the program description. The abort file should contain a sequence of Debug commands that will enable you to determine why the task failed.

PMP$EXIT

Purpose Terminates the calling task, returning the specified status record to the initiating task.

Format **PMP$EXIT (status)**

Parameter **status**: ost$status;

Status record returned to the task that initiated this task. The status record is copied to the task_status variable specified on the PMP$EXECUTE call that initiated the task.

Condition Identifier None.

Remarks The PMP$EXIT procedure is used to indicate that the task could not perform its function due to an error in the job environment or in the parameter list passed to it.

Task Communication 4

Task Communication 4

Task communication within a job is provided by two NOS/VE mechanisms, the general wait and the queue.

General Wait

The general wait mechanism is called by the OSP$AWAIT_ACTIVITY_COMPLETION call. The task is suspended until one of the specified activities completes. The possible activities include the expiration of a period of time, the completion of a task, or the receipt of a message via a queue.

The general wait of the OSP$AWAIT_ACTIVITY_COMPLETION call allows resumption of the task as the result of any of the events specified on the call. The PMP$AWAIT_TASK_TERMINATION and PMP$RECEIVE_FROM_QUEUE calls can also suspend a task but can specify only one event to resume the task.

ready_index: VAR of integer;

Index into the wait list indicating the event that occurred.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$unknown_queue_identifier
pme$usage_bracket_error

PMP$DEFINE_QUEUE

Purpose Defines a queue.

Format **PMP$DEFINE_QUEUE (name, removal_bracket, usage_bracket, status)**

Parameters **name**: pmt$queue_name;

Queue name.

removal_bracket: ost$ring;

Highest ring from which the queue definition can be removed (1 through 15). It must be greater than or equal to the ring from which the request is made.

usage_bracket: ost$ring;

Highest ring from which the queue can be used (1 through 15). It must be greater than or equal to the removal bracket ring.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$incorrect_queue_name
pme$maximum_queues_defined
pme$queue_already_defined
pme$request_gt_removal_ring
pme$usage_lt_removal_bracket

PMP$CONNECT_QUEUE

Purpose Connects the task to a queue.

Format **PMP$CONNECT_QUEUE (name, qid, status)**

Parameters

name: pmt$queue_name;

Queue name as defined by a PMP$DEFINE_QUEUE call.

qid: VAR of pmt$queue_connection;

Queue connection identifier assigned by system (1 through PMC$MAX_QUEUES_PER_JOB).

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$incorrect_queue_name
pme$maximum_tasks_connected
pme$task_already_connected
pme$unknown_queue_name
pme$usage_bracket_error

PMP$RECEIVE_FROM_QUEUE

Purpose Receives a message from a queue.

Format **PMP$RECEIVE_FROM_QUEUE (qid, wait, message, status)**

Parameters **qid**: pmt$queue_connection;

Queue connection identifier returned by the PMP$CONNECT_QUEUE call.

wait: ost$wait;

Action taken if the queue is empty.

OSC$WAIT

Suspend task until a message is received.

OSC$NOWAIT

Continue task if message is not available.

PMP$SEND_TO_QUEUE

Purpose Sends a message to a queue.

Format **PMP$SEND_TO_QUEUE (qid, message, status)**

Parameters **qid**: pmt$queue_connection;

Queue connection identifier returned by the PMP$CONNECT_QUEUE call.

message: pmt$message;

Message sent to the queue.

Field	Content
sender_id	Task identifier assigned by system (type PMT$TASK_ID).
sender_ring	Ring of task (type OST$RING, 0 through OSC$MAX_RING).
contents	Key field indicating the message pointer kind (type PMT$MESSAGE_KIND). PMC$MESSAGE_VALUE Message in value field.
value	Message sequence (type PMT$MESSAGE_VALUE).

status: VAR of ost$status;

Status record The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$error_number_of_segments
pme$error_pointer_privilege
pme$error_segment_privilege
pme$error_segment_message
pme$incorrect_message_type
pme$incorrect_segment_message
pme$maximum_queued_messages
pme$maximum_queued_segments
pme$pass_share_prohibited
pme$unknown_queue_identifier
pme$usage_bracket_error

PMP$GET_QUEUE_LIMITS

Purpose Returns the queue limits for the job.

Format **PMP$GET_QUEUE_LIMITS (queue_limits, status)**

Parameters **queue_limits**: VAR of pmt$queue_limits;

Limits record.

maximum_queues

Maximum queues that can be defined in the job (type PMT$QUEUES_PER_JOB, 0 through PMC$MAX_QUEUES_PER_JOB).

maximum_connected

Maximum tasks that can be connected to a queue (type PMT$CONNECTED_TASKS_PER_QUEUE, 0 through PMC$MAX_QUEUES_PER_JOB).

maximum_messages

Maximum messages per queue (type PMT$MESSAGES_PER_QUEUE, 0 through PMC$MAX_MESSAGES_PER_QUEUE).

status: VAR of ost$status;

Status record.

Condition Identifier None.

PMP$STATUS_QUEUES_DEFINED

Purpose Returns the number of currently defined queues.

Format **PMP$STATUS_QUEUES_DEFINED (count, status)**

Parameters **count**: VAR of pmt$queues_per_job;

Number of queues currently defined (0 to PMC$MAX_QUEUES_PER_JOB).

status: VAR of ost$status;

Status record.

Condition Identifier None.

```
{ This procedure executes the worker task with which }
{ the control task communicates.  It assumes that }
{ the control task and the worker task both reside }
{ in the same object file or object library. }

PROCEDURE execute_worker_task
  (shared_segment_name: amt$local_file_name,
  shared_segment_attributes:
    array [1 .. *] OF amt$file_item,
  communication_queue_name: pmt$queue_name;
  VAR task_id: pmt$task_id;
  VAR task_status: pmt$task_status;
  VAR status: ost$status);

  VAR
{ Parameter variables for PMP$GET_PROGRAM_SIZE. }
  number_of_object_files: pmt$number_of_object_files,
  number_of_modules: pmt$number_of_modules,
  number_of_libraries: pmt$number_of_libraries,

{ Pointer to the program description for the worker }
{ task. }
  worker_program: ^pmt$program_description,

{ Pointer to the program attributes variable in the }
{ program description. }
  worker_program_attributes: ^pmt$program_attributes,

{ Pointer to the parameter list for the worker task. }
  worker_program_parameters: ^pmt$program_parameters,

{ Pointers to parameters in the parameter list. }
  shared_segment_name_parm: ^amt$local_file_name,
  number_of_segment_attributes: ^1..amc$max_attribute,
  shared_segment_attributes_parm:
    ^array [1 .. *] of amt$file_item,
  communication_queue_name_parm: ^pmt$queue_name;
```

```
{ Builds the following worker task parameter list. }

{ 1.  Shared segment local file name.}
{ 2.  Size of the array defining the correct file }
{     attributes for opening the shared segment. }
{ 3.  File attributes for the shared segment. }
{ 4.  Name of the local queue to be used for }
{     communication between tasks. }

PUSH worker_program_parameters
  [[REP 1 OF amt$local_file_name,
  REP 1 OF amt$file_attribute_keys,
  REP UPPERBOUND (shared_segment_attributes) OF
  amt$file_item,
  REP 1 of pmt$queue_name]];
RESET worker_program_parameters;

{1} NEXT shared_segment_name_parm IN
      worker_program_parameters;
    shared_segment_name_parm^ := shared_segment_name;

{2} NEXT number_of_segment_attributes IN
      worker_program_parameters;
    number_of_segment_attributes^ :=
      UPPERBOUND (shared_segment_attributes);

{3} NEXT shared_segment_attributes_parm:
      [1 .. UPPERBOUND (shared_segment_attributes)]
      IN worker_program_parameters;
    shared_segment_attributes_parm^ :=
      shared_segment_attributes;

{4} NEXT communication_queue_name_parm
      IN worker_program_parameters;
    communication_queue_name_parm^ :=
      communication_queue_name;
```

```
{ The following variables define a message in the }
{ segment and the relative pointer to the message. }

message_to_worker: pmt$message,
message_to_worker_value_pointer: ^pmt$message_value,
worker_text_pointer: ^string (8),
worker_text_relative_ptr_ptr:
  ^rel (HEAP ( * )) ^string (8);

{ Creates the segment used to pass information }
{ between tasks. }

AMP$OPEN (shared_segment_name, amc$segment,
  ^shared_segment_attributes, shared_segment_id,
  status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Gets a heap pointer to the beginning of the segment. }

AMP$GET_SEGMENT_POINTER (shared_segment_id,
  amc$heap_pointer, shared_segment_pointer, status);
IF NOT status.normal THEN
  RETURN;
IFEND;
shared_heap := shared_segment_pointer.heap_pointer;

{ Defines and initializes the communication queue. }

PMP$DEFINE_QUEUE (communication_queue_name,
  osc$user_ring, osc$user_ring, status);
IF NOT status.normal THEN
    RETURN;
IFEND;
PMP$CONNECT_QUEUE (communication_queue_name,
  communication_queue, status);
IF NOT status.normal THEN
  RETURN;
IFEND;
```

```
{ This is the worker task program started by the }
{ control task. }

MODULE try_queues_worker_task;

??PUSH (LISTEXT := ON)??
*copyc AMP$OPEN
*copyc AMP$GET_SEGMENT_POINTER
*copyc PMP$CONNECT_QUEUE
*copyc PMP$RECEIVE_FROM_QUEUE
??POP??

PROGRAM worker_task (parameters:
  pmt$program_parameters;
  VAR status: ost$status);

  VAR
 { Pointer to the parameter list passed to the task. }

  worker_parameters: ^pmt$program_parameters,

 { These variables have the same functions as the }
 { control task variables with the same names. }

  shared_segment_name: ^amt$local_file_name,
  number_of_segment_attributes: ^1 .. amc$max_attribute,
  shared_segment_attributes:
    ^array [1 .. *] of amt$file_item,
  communication_queue_name: ^pmt$queue_name,
  communication_queue: pmt$queue_connection,
  shared_segment_id: amt$file_identifier,
  shared_segment_pointer: amt$segment_pointer,
  shared_heap: ^HEAP ( * ),
  message_from_control: pmt$message,
  worker_text_pointer: ^string (8),
  message: ^pmt$message_value,
  worker_text_relative_ptr_ptr:
    ^rel (HEAP ( * )) ^string (8);

worker_parameters := ^parameters;
RESET worker_parameters;
```

```
{ Worker task is now ready to communicate with the }
{ control task.  This call requests a message from }
{ the queue and waits until a message is available. }

PMP$RECEIVE_FROM_QUEUE (communication_queue, osc$wait,
  message_from_control, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Initialize a sequence pointer to access the queue }
{ message. }

message := ^message_from_control.value;
RESET message;

{ Get the relative pointer to the item in the shared }
{ segment from the message passed on the local queue. }

NEXT worker_text_relative_ptr_ptr IN message;

{ Build a direct pointer from the relative pointer }
{ and the pointer to the shared segment. }

worker_text_pointer :=
  #PTR (worker_text_relative_ptr_ptr^, shared_heap^);
IF worker_text_pointer^ = 'Hello!' THEN
{ Rejoice! Rejoice! Rejoice greatly! }
IFEND;

PROCEND worker_task;
MODEND try_queues_worker_task;
```

Condition Processing 5

Condition Processing 5

A condition is an event that interrupts normal task processing. Conditions are grouped into the following categories.

System conditions.

Block exit processing conditions.

Interactive conditions.

Job resource conditions.

Segment access conditions.

Process interval timer condition.

User-defined conditions.

This chapter describes the following topics.

- Enabling and disabling detection of system conditions.
- Processing of conditions that occur within a task.

System Condition Detection

The PMP$ENABLE_SYSTEM_CONDITIONS and PMP$INHIBIT_SYSTEM_CONDITIONS procedures enable and disable, respectively, the detection of a set of system conditions. Table 5-1 lists the system conditions that can be specified on the calls. It also indicates whether the condition is enabled or disabled when the task begins.

If a system condition occurs while detection of the condition is disabled, the condition remains pending. If the task subsequently enables detection of the condition, NOS/VE clears the pending condition before enabling its detection.

Table 5-1. System Conditions That Can Be Enabled or Disabled

Identifier	Initial State
PMC$ARITHMETIC_OVERFLOW	Enabled
PMC$ARITHMETIC_SIGNIFICANCE	Enabled
PMC$DIVIDE_FAULT	Enabled
PMC$EXPONENT_OVERFLOW	Enabled
PMC$EXPONENT_UNDERFLOW	Enabled
PMC$FP_INDEFINITE	Enabled
PMC$FP_SIGNIFICANCE_LOSS	Disabled
PMC$INVALID_BDP_DATA	Enabled

PMP$INHIBIT_SYSTEM_CONDITIONS

Purpose Disables detection of the specified system conditions.

Format **PMP$INHIBIT_SYSTEM_CONDITIONS (conditions, status)**

Parameters **conditions**: pmt$system_conditions;

Condition set inhibited. The set can contain any of the following identifiers.

PMC$ARITHMETIC_OVERFLOW

Arithmetic overflow.

PMC$ARITHMETIC_SIGNIFICANCE

Arithmetic significance loss.

PMC$DIVIDE_FAULT

Divide fault.

PMC$EXPONENT_OVERFLOW

Floating point exponent overflow.

PMC$EXPONENT_UNDERFLOW

Floating point exponent underflow.

PMC$FP_INDEFINITE

Floating point indefinite.

PMC$FP_SIGNIFICANCE_LOSS

Floating point significance loss.

PMC$INVALID_BDP_DATA

Invalid BDP data.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifier pme$unselectable_condition

Table 5-2. Condition Processing When No Condition Handler Is in Effect

Category	System Standard Processing
System condition	Returns abnormal status and aborts the task.
Block exit condition	No processing; the task resumes.
Interactive condition	Asks the interactive user whether the task should resume or terminate.
Job resource condition	Within an interactive job, asks the interactive user whether the limit should be increased. Within a batch job, returns abnormal status and aborts the job.
Segment access condition	Returns abnormal status and aborts the task.
Process interval timer condition	No processing; the task resumes.
User-defined condition	No processing; the task resumes.

Table 5-3. Condition Handler Scope

Condition Category	Scope
System conditions	Establishing block and its subordinate blocks within the same execution ring.
Block exit processing conditions	Establishing block only.
Interactive conditions	Establishing block and its subordinate blocks.
Job resource conditions	Establishing block and its subordinate blocks.
Segment access conditions	Establishing block and its subordinate blocks within the same execution ring.
Process interval timer condition	Establishing block and its subordinate blocks.
User-defined conditions	Establishing block and its subordinate blocks within the same execution ring.

Table 5-4. Condition Set Specification

Selector Identifier	Condition Field Name	Condition Identifiers
PMC$ALL_CONDITIONS	None.	None.
PMC$CONDITION_ COMBINATION	combination	Set of category identifiers.
PMC$SYSTEM_CONDITIONS	system_conditions	Set of one or more condition identifiers listed in table 5-5 (PMT$SYSTEM_CONDITIONS).
PMC$BLOCK_EXIT_ PROCESSING	reason	Set of one or more of the following condition identifiers: PMC$BLOCK_EXIT Either a nonlocal EXIT statement was executed, deactivating the block, or the procedure completed and control returned to the procedure that called it. PMC$PROGRAM_ TERMINATION A PMP$EXIT call was executed. PMC$PROGRAM_ABORT A PMP$ABORT call was executed.
JMC$JOB_RESOURCE_ CONDITION	job_resource_condition	JMC$TIME_LIMIT_CONDITION Approaching time limit.
MMC$SEGMENT_ ACCESS_CONDITION	segment_access_ condition_identifier	Only one of the following condition identifiers: MMC$SAC_READ_ BEYOND_EOI Read beyond highest page accessed. MMC$SAC_READ_WRITE_ BEYOND_MSL Read or write beyond the maximum segment length. MMC$SAC_IO_READ_ERROR Read or write error on backup disk storage.
IFC$INTERACTIVE_ CONDITION	interactive_condition	Only one of the following condition identifiers: IFC$PAUSE_BREAK The interactive user interrupted the task. IFC$TERMINATE_BREAK The interactive user terminated the task.
PMC$PIT_CONDITION	None.	None.
PMC$USER_DEFINED_ CONDITION	user_condition_name	User-defined condition name.

PMP$ESTABLISH_CONDITION_HANDLER

Purpose Specifies condition handler procedure to process the specified conditions.

Format **PMP$ESTABLISH_CONDITION_HANDLER (conditions, condition_handler, establish_descriptor, status)**

Parameters

conditions: pmt$condition;

Condition set the procedure processes (see table 5-4).

condition_handler: pmt$condition_handler;

Pointer to the condition handler procedure.

establish_descriptor: ^pmt$established_handler;

Pointer to descriptor space allocated within the current stack frame.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$descriptor_address_error
pme$handler_stack_error
pme$inconsistent_stack
pme$incorrect_condition_name
pme$invalid_condition_selector
pme$stack_overwritten
pme$unselectable_condition

PMP$DISESTABLISH_COND_HANDLER

Purpose Disestablishes the condition handler currently in effect for the specified conditions.

Format **PMP$DISESTABLISH_COND_HANDLER (conditions, status)**

Parameters **conditions**: pmt$condition;

Condition set specified on the PMP$ESTABLISH_COND_HANDLER call that established this condition handler (see table 5-4).

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers
pme$handler_stack_error
pme$incorrect_condition_name
pme$inconsistent_stack
pme$no_established_handler

Remarks A condition handler can only be disestablished within its scope. The scope of a condition handler depends on its condition category as shown in table 5-3.

System Condition Handler

A system condition handler can be established only for selectable system conditions. Table 5-1 lists the selectable system conditions.

A task can establish a condition handler for a system condition while detection of the condition is inhibited. The condition handler is not used until detection of the condition is enabled. Any pending condition is cleared before detection of the condition is enabled.

If the system condition occurs within the condition handler for that condition and the condition handler has not established a new condition handler for the condition, NOS/VE terminates the task and returns abnormal status.

If a system condition occurs while a condition handler is in effect for the condition, NOS/VE passes a pointer to the stack frame save area of the block where the system condition occurred. With the following exceptions, the P register in the stack frame save area points to the instruction that caused the system condition. In the exceptions, the P register points to the instruction that follows the instruction that caused the system condition.

Ring number zero	Exponent underflow
Exponent overflow	Floating point significance loss

PMP$CONTINUE_TO_CAUSE

Purpose

Continues the condition, causing NOS/VE to call the next most recently established condition handler in effect for the condition. The condition must be within the scope of the condition handler.

Format

PMP$CONTINUE_TO_CAUSE (standard, status)

Parameters

standard: pmt$standard_selection;

Indicates whether or not NOS/VE should call the system standard procedure if no other condition handler is in effect for the condition.

PMC$EXECUTE_STANDARD_PROCEDURE

Call the system standard procedure.

PMC$INHIBIT_STANDARD_PROCEDURE

Do not call the system standard procedure; return abnormal status to the condition handler that issued the call.

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifers

pme$handler_stack_error
pme$inconsistent_stack
pme$invalid_condition_handler
pme$invalid_standard_selection
pme$no_condition_to_continue
pme$no_established_handler
pme$recursive_continue
pme$stack_overwritten

Remarks

If no other condition handler is in effect for the condition and the call specifies PMC$EXECUTE_STANDARD_PROCEDURE, NOS/VE executes the standard condition processing procedure for the condition category. Standard condition processing for each condition category is described in table 5-2.

Interactive Condition Handler

An interactive condition occurs when an interactive user interrupts his or her terminal session with a pause break or a terminate break.

A PMP$ESTABLISH_CONDITION_HANDLER call can associate only one interactive condition with a condition handler. Therefore, to associate multiple interactive conditions with a condition handler, the task must issue a PMP$ESTABLISH_CONDITION_HANDLER call for each condition.

The following are possible ways that a condition handler could process an interactive condition.

- Prompt the interactive user for direction.
- Circumvent the interrupted process via a nonlocal exit.
- Return normal status, allowing the task to resume.
- Return abnormal status to terminate the task.
- Call PMP$EXIT or PMP$ABORT to terminate the task.

Job Resource Condition Handler

A job resource condition warns the task of an impending time limit violation.

A PMP$ESTABLISH_CONDITION_HANDLER call can associate only one job resource condition with a condition handler. (Currently, only one job resource condition, time limit, exists.)

The following are possible ways that a condition handler could process a job resource condition.

- Increase the limit associated with the condition and return normal status.
- Return abnormal status to terminate the task.
- Call PMP$EXIT or PMP$ABORT to terminate the task.

Process Interval Timer Condition Handler

The process interval timer condition notifies the task of the expiration of the process interval timer. The condition occurs only when the following qualifications are met.

- A process interval timer condition handler is in effect.
- The process interval timer for the task has been set by a PMP$SET_PROCESS_INTERVAL_TIMER call.
- The process interval timer decrements to zero.

The process interval timer decrements only while the task is actually using the central processor; it does not decrement while the processor has interrupted to monitor mode or has been dispatched to another task.

User-Defined Condition Handler

A task can define a condition by naming it on a PMP$ESTABLISH_CONDITION_HANDLER call. The user-defined condition occurs when the task specifies the condition on a PMP$CAUSE_CONDITION call.

A PMP$ESTABLISH_CONDITION_HANDLER call can associate only one user-defined condition with a condition handler. Therefore, to associate multiple user-defined conditions with a condition handler, the task must issue a PMP$ESTABLISH_CONDITION_HANDLER call for each condition.

If the task calls the cause_condition procedure while no condition handler is in effect for the condition, the procedure returns abnormal status and the task resumes.

If the task calls the cause_condition procedure while a condition handler is in effect for the condition, NOS/VE passes the descriptor pointer specified on the cause_condition call to the condition handler.

The following are possible ways that a condition handler could process a user-defined condition.

- Resume the task by returning normal status.
- Terminate the task by returning abnormal status or by calling the PMP$EXIT or PMP$ABORT procedures.

If the condition handler returns normal status, the task resumes using the same stack frame information in use when the condition occurred.

PMP$TEST_CONDITION_HANDLER

Purpose

Simulates the occurrence of an error condition to allow testing of a condition handler for those conditions.

Format

PMP$TEST_CONDITION_HANDLER (conditions, save_area, status)

Parameters

conditions: pmt$condition;

Condition to be forced (see table 5-4).

save_area: ^ost$stack_frame_save_area;

Stack frame save area image to be passed to the condition handler (see appendix D).

status: VAR of ost$status;

Status record. The process identifier returned is PMC$PROGRAM_MANAGEMENT_ID.

Condition Identifiers

pme$handler_stack_error
pme$inconsistent_stack
pme$invalid_condition_handler
pme$no_established_handler
pme$unsupported_by_test_cond

Remarks

The condition can be a job resource, segment access, interactive, or process interval timer condition or a set of system conditions; it cannot be a block exit or user-defined condition or all conditions or a combination of condition categories.

Message Generation 6

Message Generation 6

The NOS/VE program interface includes procedures to generate standard status records and error messages. A standard status record describes one of the system-defined exception conditions listed in the Diagnostic Messages manual. The manual also lists the standard message templates associated with the conditions.

Status Record Generation

As described in chapter 1, a procedure returns a status record to indicate its completion status. The status record must be of type OST$STATUS. To indicate normal completion, the normal field of the status record is set to TRUE. To indicate abnormal completion, you must initialize abnormal status in the status record.

To initialize an abnormal status record, the task can call the following procedures.

- OSP$SET_STATUS_ABNORMAL for general status record initialization.
- AMP$SET_FILE_INSTANCE_ABNORMAL for status record initialization according to file interface conventions when the file identifier is known. (This procedure is described in an appendix on file access procedures in the CYBIL File Interface manual.)
- OSP$SET_STATUS_FROM_CONDITION for status record initialization within a condition handler. The status record generated is for the condition that caused the system to call the condition handler.

OSP$SET_STATUS_ABNORMAL

Purpose Initializes an abnormal status record.

Format **OSP$SET_STATUS_ABNORMAL (identifier, condition, text, status)**

Parameters

identifier: string (2);

Two-character identifier of the process that detected the condition.

condition: ost$status_condition;

Exception condition (specified by a condition identifier or the integer code for the condition, 0 through OSC$MAX_CONDITION).

text: string (*);

String to be used as the first status parameter in the text field.

status: VAR of ost$status;

Initialized status record.

Condition Identifier None.

Remarks

- If the specified text string is not the null string (its length is nonzero), OSP$SET_STATUS_ABNORMAL inserts the string into the status record text field as the first status parameter. The first character of the text field is set to the OSC$STATUS_PARAMETER_DELIMITER character.

  The OSP$FORMAT_MESSAGE procedure uses the first character of the text field, OSC$STATUS_PARAMETER_DELIMITER character, to determine the beginning of each status parameter in a status record.

- OSP$SET_STATUS_ABNORMAL discards any trailing space characters in the string specified as the text parameter before appending the string to the status record text field.

- If the text string (after trailing spaces are discarded) does not fit in the status record text field, OSP$SET_STATUS_ABNORMAL truncates the rightmost characters so that the string will fit into the field.

OSP$APPEND_STATUS_INTEGER

Purpose

Converts an integer to its string representation and appends the string to the text field of the status record.

Format

OSP$APPEND_STATUS_INTEGER (delimiter, int, radix, include_radix_specifier, status)

Parameters

delimiter: char;

First character appended. If the character matches the first character of the text field (OSC$STATUS_PARAMETER_DELIMITER), the text becomes the next status parameter; if it does not match, the text is appended to the previous status parameter.

int: integer;

Integer value.

radix: 2 .. 16;

Specifies the radix for the integer parameter (2 through 16).

include_radix_specifier: boolean;

Specifies whether to include the radix representation in the string.

TRUE

Include radix.

FALSE

Omit radix.

status: VAR of ost$status; [input, output]

Status record to which the integer is appended.

Condition Identifier

None.

Remarks

- OSP$APPEND_STATUS_INTEGER discards any trailing space characters in the string specified as the text parameter before appending the string to the status record text field.
- If the text string (after trailing spaces are discarded) does not fit in the status record text field, OSP$APPEND_STATUS_INTEGER truncates the rightmost characters so that the string is the correct length for the field.

Status Severity Check

After calling a procedure, a task must check the status record returned. It must first determine whether the status returned is normal or abnormal; for example, you can use the following first phrase of an IF statement.

```
IF NOT status.NORMAL THEN
```

If the status returned is abnormal, it can then, if appropriate, check the status severity level by calling OSP$GET_STATUS_SEVERITY as illustrated in figure 6-1.

```
{ This module contains CYBIL procedures to generate a message }
{ and output it to the caller's job log if the completion of }
{ procedure PMP$LOAD returns a status condition greater }
{ in severity than OSC$WARNING. }

MODULE sample;

{ Required *COPYC directives to use CYBIL procedures }
{ in the CYBIL module. }
*copyc pmp$load;
*copyc pmp$log;
*copyc osp$format_message
*copyc osp$get_status_severity;

    PROCEDURE [XDCL] sample (entry_name: pmt$program_name);

    CONST
      max_line_size = 60;

    VAR
      entry_address: pmt$loaded_address,
      severity: ost$status_severity,
      message: ost$status_message,
      pointer: ^ost$status_message,
      msg_line_count: ^ost$status_message_line_count,
      msg_line_size: ^ost$status_message_line_size,
      msg_line_text: ^string (*),
      i: 1 .. osc$max_status_message_lines,
      stat: ost$status,
      ignore_status: ost$status;
```

(Continued)

Figure 6-1. Checking the Status Severity Level

OSP$GET_STATUS_SEVERITY

Purpose Returns the severity level of the status condition.

Format **OSP$GET_STATUS_SEVERITY (condition, severity, status)**

Parameters **condition**: ost$status_condition;

Condition code (condition field from status record).

severity: VAR of ost$status_severity;

Severity level.

OSC$INFORMATION_STATUS

Informative status.

OSC$WARNING_STATUS

Warning status.

OSC$ERROR_STATUS

Error status.

OSC$FATAL_STATUS

Fatal status.

OSC$CATASTROPHIC_STATUS

Catastrophic status.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Message Levels

An OSP$FORMAT_MESSAGE call specifies the message level of the generated message. The message level is the level of detail of the message. The following are the message levels.

Brief
: Error message without the process identifier and condition code. If a path is inserted in the message, it is presented relative to the working catalog; standard file names appear without the $LOCAL prefix.

Full
: Error message with the process identifier and condition code. If a path is inserted in the message, it is presented as an absolute path.

Explain
: Currently the same as full mode.

For example, the following is the brief message for condition identifier PME$MAXIMUM_QUEUED_MESSAGES.

```
--ERROR-- Maximum number of messages are already on MY_QUEUE.
```

The following is the full message.

```
--ERROR PM 235061-- Maximum number of messages are already
on MY_QUEUE.
```

The message level displayed in the job can be set by the SET_MESSAGE_MODE command described in the SCL System Interface manual. A task can determine the current message level with an OSP$GET_MESSAGE_LEVEL call and change the current message level with an OSP$SET_MESSAGE_LEVEL call.

Remarks

- If a message template is defined for the specified condition, the status parameters within the text string are inserted in the template to form the status message.
- If no message template is defined for the specified condition, OSP$FORMAT_MESSAGE returns the contents of the status record within the following line.

```
ID=xx CC=code TEXT=string
```

- If the generated message is longer than the specified max_message_line parameter value, OSP$FORMAT_MESSAGE splits the message into more than one line so that no line is longer than the maximum length. It attempts to split the message at a delimiter. If that is not possible, it appends two periods to the end of the line to indicate continuation on the next line.
- Any character in the inserted text that cannot be printed is represented in the formatted message by a question mark (?) character.
- The example in figure 6-1 illustrates the use of OSP$FORMAT_MESSAGE.

OSP$SET_MESSAGE_LEVEL

Purpose Sets the message level of the job.

Format **OSP$SET_MESSAGE_LEVEL (message_level, status)**

Parameters **message_level**: ost$status_message_level;
Current message level setting.

OSC$CURRENT_MESSAGE_LEVEL
Current.

OSC$BRIEF_MESSAGE_LEVEL
Brief.

OSC$FULL_MESSAGE_LEVEL
Full.

OSC$EXPLAIN_MESSAGE_LEVEL
Currently, the same as full mode.

status: VAR of ost$status;
Status record.

Condition Identifier None.

Interstate Communications 7

Interstate Communications 7

NOS jobs and NOS/VE jobs can be executed simultaneously. A task within a NOS/VE job can start a NOS job. After starting the NOS job, the NOS/VE task can send data to and receive data from the job.

The task and job communicate via a link file that acts as a message buffer. The NOS/VE task starts the NOS job by opening the link file. After it closes the link file, the NOS/VE task can no longer communicate with the NOS job, although the NOS job continues until its termination.

A NOS/VE task can have only one link file open at a time. Therefore, although a NOS/VE task can start more than one NOS job before it terminates, it can communicate with only one NOS job at a time.

The only NOS/VE task with which the started NOS job can communicate is the NOS/VE task that started it.

Creating a NOS Job

To start a NOS job, a NOS/VE task must prepare a NOS job command record and a link file. To prepare a NOS job command record, the NOS/VE task performs the following steps.

1. Declares a NOS/VE command variable.
2. Stores the NOS command record in the NOS/VE command variable.
3. Executes the SCL command SET_LINK_ATTRIBUTES to specify NOS job accounting information.

To prepare a link file, the NOS/VE job performs the following steps:

1. Requests a link file.
2. Stores the NOS/VE command variable name as the user_info attribute value for the link file.

Requesting a link file can only be performed using the SCL command REQUEST_LINK. However, the user_info attribute value can be set by either an SCL command or a CYBIL procedure call.

Command Record Size Limit

The system cannot pass a NOS command record longer than 508 60-bit words. If the command record is too large, the AMP$OPEN call to open the link file returns abnormal status (ICE$PARTNER_JOB_TOO_LONG).

The size limit includes the Z record delimiters the system adds when it converts the NOS commands to NOS 64-character set Z records.

To minimize the length of the NOS/VE command variable, store the NOS command sequence for the job as a NOS procedure. The NOS commands in the NOS/VE command variable would then consist only of those NOS commands required to access and execute the procedure file.

For example, the following NOS/VE commands declare a string variable named NOSJOB and store a NOS command record that calls a NOS procedure in the string variable.

```
CREATE_VARIABLE NAME=nos_job_record KIND=string
nos_job_record = 'myjob.;get,nosproc.;begin,,nosproc.'
```

The string variable could also be dimensioned as in the following example.

```
CREATE_VARIABLE NAME=nos_job_record KIND=string..
DIMENSION=1..3
nos_job_record(1) = 'myjob.'
nos_job_record(2) = 'get,nosproc.'
nos_job_record(3) = 'begin,,nosproc.'
```

By dimensioning the string variable, you can use semicolons in the NOS commands. However, the variable DIMENSION must be more than a single entry. (It cannot be 1..1, 2..2, and so forth.)

Note that each NOS command in the example ended with a period. Also, note that the NOS commands were in lowercase letters. This is valid because the system converts all lowercase characters assigned to the command variable to uppercase characters. Any trailing blanks are suppressed. The ASCII character codes are converted to six-bit display codes for use by the NOS system. Any ASCII character that cannot be converted to display code becomes an asterisk.

Assigning Link File Attributes

The NOS/VE job or task must set the user_info attribute value for the link file. The attribute value must be the name of the command variable containing the NOS command record.

The REQUEST_LINK command sets the FAP attribute of the link file to the name of the interstate communication FAP. Therefore, you cannot associate another FAP with a link file.

Specifying values for the following file attributes affects link file processing. Values can be specified for other file attributes, but link file processing does not use the values.

access_mode

To send information from the NOS job to the NOS/VE task, the access_mode attribute must include the PFC$READ value. To send information from the NOS/VE task to the NOS job, the access_mode attribute must include PFC$SHORTEN, PFC$APPEND, or PFC$MODIFY values. All other access_mode values are ignored.

error_exit_name

It can specify an error processing routine for the link file.

file_organization

It must be AMC$SEQUENTIAL.

return_option

It indicates whether the link file is detached when the file is closed or when the job terminates.

ring_attributes

Because the link file cannot be executed, only the read and write brackets are relevant.

As described in the SCL System Interface and CYBIL File Interface manuals, the following commands and procedures can set file attribute values.

SET_FILE_ATTRIBUTES command

CHANGE_FILE_ATTRIBUTES command

AMP$FILE call

AMP$OPEN call

Communication Between the Task and Job

The NOS/VE task and the started NOS job communicate by reading and writing data to the link file. The link file acts as a message buffer.

If the NOS/VE task attempts to read or write data on the link file and the NOS job has not yet opened the link file, the NOS/VE task is suspended. To determine if the NOS job has opened the link file before attempting to read or write to the file, the task can call AMP$FETCH_ACCESS_INFORMATION and check whether the link file last_op_status is AMC$COMPLETE.

Link File Deadlock

The NOS/VE task and the NOS job must not be both reading or both writing to the link file at the same time. If they do, ICF detects a deadlock and returns abnormal status (ICE$READ_DEADLOCK or ICE$WRITE_DEADLOCK) to the NOS/VE task.

To clear a read deadlock condition (ICE$READ_DEADLOCK), the task must perform a write operation (such as an AMP$PUT_NEXT call). Similarly, to clear a write deadlock condition (ICE$WRITE_DEADLOCK), the task must perform a read operation (such as an AMP$GET_NEXT call).

Data Conversion

The system does not convert link file data. NOS/VE uses a 64-bit word; NOS uses a 60-bit word. Therefore, when NOS/VE passes data to a NOS job, the first four bits of each eight-byte word are lost, and when NOS passes data to the NOS/VE task, the first four bits of each eight-byte word are zero.

The NOS/VE task must arrange the data in each word so that the data passed to the NOS job is meaningful. The task may perform data conversion using a FAP associated with a file other than the link file. Data would be accessed via the file associated with the FAP and then converted and transferred to and from the link file by the FAP.

Fetching Information About the Link File

The following file access information items returned by an AMP$FETCH_ACCESS_INFORMATION call are meaningful for a link file.

error_status

Returns the condition code returned by the last file interface request.

file_position

A file_position of AMC$EOP indicates that the NOS job has sent a partition delimiter.

last_access_operation

Returns the last access request issued for this instance of open.

last_op_status

A last_op_status of AMC$COMPLETE indicates that the NOS job has opened the link file.

previous_record_length

Returns the number of bytes in the last full record accessed.

Positioning the Link File

An AMP$REWIND call for a link file resets the file position to AMC$BOI. The AMP$SKIP call is not supported for link files.

Unsupported File Interface Calls

The operations performed by the following file interface calls are undefined for a link file. Therefore, when a link file is specified on one of these calls, the procedure returns normal status but does not perform the requested operation.

AMP$SEEK_DIRECT
AMP$SKIP

The following file interface calls are invalid for a link file.

AMP$GET_SEGMENT_POINTER
AMP$SET_SEGMENT_EOI
AMP$SET_SEGMENT_POSITION
AMP$WRITE_TAPE_MARK

NOS Link Subroutines

The following are the subroutines used to read and write to a link file.

Procedure	Function
OPENLNK	Opens the link file.
CLOSLNK	Closes the link file.
GETNLNK	Reads a record from the link file.
GETPLNK	Reads a partial record from the link file.
PUTNLNK	Writes a record on the link file.
PUTPLNK	Writes partial record on the link file.
WREPLNK	Writes a partition delimiter on the link file.

The subroutine descriptions follow in the order the subroutines are listed above.

OPENLNK Subroutine

The OPENLNK subroutine opens the link file for reading and writing by the NOS job step.

NOTE

If a NOS job not started by a NOS/VE task calls the OPENLNK subroutine, the system aborts the job without reprieve or exit processing.

The subroutine call has the following format.

CALL OPENLNK (status)

status

Name of variable into which the subroutine returns one of the following integer condition codes.

0 Normal completion.

1 The link file is already open.

2 The NOS/VE task has closed the link file.

GETNLNK Subroutine

The GETNLNK subroutine reads the next record of data from the link file.

The read always begins at the beginning of the next record.

If the working storage area is not long enough for the entire record, the subroutine fills the working storage area and sets the file position as midrecord. The job step must call the GETPLNK subroutine to get the rest of the record.

The subroutine call has the following format.

CALL GETNLNK (wsa, wsal, length, unused, position, status)

wsa

Name of the working storage area.

wsal

Name of the variable containing the length of the working storage area.

length

Name of the variable in which the integer number of words read is returned.

unused

Name of the variable in which the unused bit count is returned. The unused bit count is the number of least significant bits in the last used working storage word that do not contain data.

position

Name of the variable in which one of the following integer position codes is returned.

1 Midrecord

2 End-of-record

3 End-of-partition

4 End-of-information

status

Name of the variable into which the subroutine returns one of the following integer condition codes.

0 Normal completion.

1 The job has not opened the link file.

2 The NOS/VE task has closed the link file.

3 The program attempted to read data after the EOI of the file.

PUTNLNK Subroutine

The PUTNLNK subroutine writes the next record of data to the link file. If the current file position is midrecord, the preceding partial record is terminated before the next record is written.

The subroutine call has the following format.

CALL PUTNLNK (wsa, wsal, status)

wsa

Name of the working storage area.

wsal

Name of the variable containing the length of the working storage area.

status

Name of the variable into which the subroutine returns one of the following integer condition codes.

0 Normal completion.

1 The job has not opened the link file.

2 The NOS/VE task has closed the link file.

PUTPLNK Subroutine

The PUTPLNK subroutine writes a partial record of data to the link file.

The subroutine can write the beginning, middle, or end of a record, depending on the value of the term parameter.

If the link file is closed before the end of a record is written, the incomplete record is terminated before the link file is closed.

The subroutine call has the following format.

CALL PUTPLNK (wsa, wsal, term, status)

wsa

Name of the working storage area.

wsal

Name of the variable containing the length of the working storage area.

Interstate Communication Example

The following example demonstrates interstate communication using these steps.

1. A CYBIL program named NOS_READ starts a NOS job.
2. The NOS job executes a NOS procedure file named PROCFIL.
3. The NOS procedure compiles and executes a FORTRAN program named VEWRITE.
4. The VEWRITE program reads a file named DATAFL and writes its data to the link file.
5. The NOS_READ program reads the data from the link file and writes it to the output file.

The following is a source listing of the INTERSTATE_EXAMPLE program.

```
MODULE interstate_example;

*copyc clp$create_variable
*copyc clp$write_variable
*copyc amp$open
*copyc amp$get_next
*copyc amp$put_next
*copyc amp$close
*copyc amp$fetch_access_information
*copyc pmp$exit

PROGRAM nos_read;

  CONST
{ This is the number of words in the array read }
{ from the link file. }
{ num_words = 25; }

  TYPE
{ The following data structure describes an array }
{ of NOS display code words.  The first four bits }
{ of each word are zero bits added when a NOS word }
{ is transferred to NOS/VE.  The rest of the word }
{ is 10 6-bit characters. }
```

```
{ The ASCII character codes in this array are }
{ ordered to correspond to the display code }
{ collating sequence. Note, however, that the }
{ ASCII code in the 00 position is 20, the code for }
{ space, rather than 3a, the ASCII code for colon. }

 ascii : [READ] ARRAY [0 .. 63] OF 0 .. 255 :=
  [20(16),41(16),42(16),43(16),44(16),45(16),46(16),47(16),
   48(16),49(16),4a(16),4b(16),4c(16),4d(16),4e(16),4f(16),
   50(16),51(16),52(16),53(16),54(16),55(16),56(16),57(16),
   58(16),59(16),5a(16),30(16),31(16),32(16),33(16),34(16),
   35(16),36(16),37(16),38(16),39(16),2b(16),2d(16),2a(16),
   2f(16),28(16),29(16),24(16),3d(16),20(16),2c(16),2e(16),
   23(16),5b(16),5d(16),25(16),22(16),5f(16),21(16),26(16),
   27(16),3f(16),3c(16),3e(16),40(16),5c(16),5e(16),3b(16)];

{ Loop that stores an ASCII code in the string to }
{ correspond to each display code in the array. }
string_position := 0;
/word_loop/
FOR word_position := 1 TO num_words DO
 /char_loop/
 FOR char_position := 1 TO 10 DO
   string_position := string_position + 1;
   CASE char_position OF
   =1=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_1]);
   =2=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_2]);
   =3=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_3]);
   =4=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_4]);
   =5=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_5]);
```

```
{ The main program begins here. }

  VAR
  status: ost$status,

  link_file: [STATIC] amt$local_file_name :=
    'LINK_FILE',
  link_fid: amt$file_identifier,
  link_access_selections: [STATIC] array [1..2] of
   amt$access_selection :=
   [[amc$user_info,'NOS_JOB_RECORD'],
   [amc$file_access_procedure,'icp$fap_control']],
{ ICP$FAP_CONTROL is the interstate communication }
{ FAP.  It can also be assigned using a }
{ REQUEST_LINK command. }

{ The following are the variable declarations used to }
{ initialize the command variable. }

  nos_job_record: clt$variable_reference,
  variable_value: clt$variable_value,
  variable_scope: clt$variable_scope,
  job_record: record
   CASE 1..2 OF
   =1=
    cv: ARRAY [1..(1*(2+256))] of cell,
   =2=
    sv: ARRAY [1..1] of
     RECORD
      size: ost$string_size,
      value: string(256),
     RECEND,
   CASEND,
  recend,

  access_info: [STATIC] array [1..1] of
   amt$access_info := [[*, amc$last_op_status, *]],

  wsa_ptr : ^CELL,
  wsa_length : amt$working_storage_length,
  word_array: word_array_type,
  string_variable: string(256),

  transfer_count: amt$transfer_count,
  byte_address: amt$file_byte_address,
  file_position: amt$file_position,
  notify: string(8);
```

```
{ These statements notify the NOS job that the NOS/VE }
{ task is ready to receive the data. }
 notify := ' READY  ';
 amp$put_next(link_fid, ^notify,8,byte_address, status);
 IF NOT status.NORMAL THEN
  pmp$exit(status);
 IFEND;

 print_string(' Sent READY to link file.');

{ The following statements read a 25-word array of }
{ data from the link file. }
 wsa_ptr := ^word_array;
 wsa_length := #size(word_array);
 amp$get_next (link_fid, wsa_ptr, wsa_length,
  transfer_count, byte_address, file_position,status);
 IF NOT status.NORMAL THEN
  pmp$exit(status);
 IFEND;

{ The following statements convert and write the data }
{ to the output file. }
 print_string(' Read the following record from link.');
 string_variable := ' ';
 convert_display_code_to_ascii(word_array,
  string_variable);
 print_string(string_variable);

{ These statements notify the NOS job that the NOS/VE }
{ task has finished reading data from the link file. }
 notify := ' DONE   ';
 amp$put_next(link_fid, ^notify,8,byte_address, status);
 IF NOT status.NORMAL THEN
  pmp$exit(status);
 IFEND;

 amp$close (link_fid, status);
 IF NOT status.NORMAL THEN
  pmp$exit(status);
 IFEND;

 amp$close(output_fid, status);
 IF NOT status.NORMAL THEN
  pmp$exit(status);
 IFEND;

 PROCEND nos_read;
 MODEND interstate_example;
```

```
      CALL GETNLNK(MESS, 1, LEN, UNUSED, POS, STATUS)
      IF (STATUS .NE. 0) GO TO 40

      CALL PUTNLNK (WSA, N, STATUS)
      IF (STATUS .NE. 0) GO TO 40
C DATA WRITTEN TO LINK FILE

      CALL GETNLNK(MESS, 1, LEN, UNUSED, POS, STATUS)
      IF (STATUS .NE. 0) GO TO 40
C DATA READ FROM LINK FILE

      CALL CLOSLNK (STATUS)
   40 STOP
      END
```

The data file read by the VEWRITE program can contain up to 25 lines of data with 10 uppercase characters per line. Assume that the following is the contents of DATAFL for this demonstration.

```
THIS IS T
HE MESSAGE
 TO BE SEN
T TO NOS/V
E
```

Assuming the NOS_READ program is stored as deck X on source library X, the following NOS/VE command sequence expands, compiles, and executes the program.

```
/scu b=x
sc/expd d=x ab=$system.cybil.osf$program_interface
sc/end no
/cybil i=compile l=listing
/lgo
Output file opened.
Command variable created.
Command variable written.
Link file opened.
NOS job returned AMC$COMPLETE status.
Sent READY to link file.
Read the following record from link.
THIS IS THE MESSAGE TO BE SENT TO
NOS/VE
/
```

Command Language Services 8

Command Language Services 8

This chapter describes procedures that provide the following system command language (SCL) services.

- Command language variable use
- String conversion

SCL uses these procedures when processing commands that specify command variables or request string conversion.

Command Language Variables

The CLP$CREATE_VARIABLE call creates a command language variable. A command language variable associates a name with a value in memory. Besides a name and a value, a variable also has a kind, a dimension, and a scope.

Variable Kind and Dimension

A variable can be any of the following kinds.

- String
- Integer
- Boolean
- Status record

The variable could also be an array of elements of the specified kind. The CLP$CREATE_VARIABLE call specifies the upper and lower bounds of the array.

A variable is initialized according to its type.

- String: null string
- Integer: zero
- Boolean: FALSE
- Status record: normal status

CLP$CREATE_VARIABLE

Purpose Declares and initializes a command language variable.

Format **CLP$CREATE_VARIABLE (name, kind, max_string_size, lower_bound, upper_bound, scope, variable, status)**

Parameters

name: string (*);

Variable name.

kind: clt$variable_kinds;

Variable kind.

CLC$STRING_VALUE

String

CLC$INTEGER_VALUE

Integer

CLC$BOOLEAN_VALUE

Boolean

CLC$STATUS_VALUE

Status record

max_string_size: ost$string_size;

Maximum length of a string variable.

lower_bound: clt$variable_dimension;

Smallest subscript of an array variable (CLC$MIN_VARIABLE_DIMENSIONS through CLC$MAX_VARIABLE_DIMENSION).

upper_bound: clt$variable_dimension;

Largest subscript of an array variable (CLC$MIN_VARIABLE_DIMENSION through CLC$MAX_VARIABLE_DIMENSION).

CLP$DELETE_VARIABLE

Purpose Removes a command variable from the current block.

Format **CLP$DELETE_VARIABLE (name, status)**

Parameters

name: string (*);

Variable name defined when the variable was declared.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Table 8-2. Variable Value (CLT$VARIABLE_VALUE)

Field	Content
descriptor	Name of the value kind as defined when the variable was created (string of length OSC$MAX_NAME_SIZE, 31 characters). When writing a variable value, you need not initialize this field.
kind	Key field identifying the value kind (CLT$VARIABLE_KINDS). CLC$STRING_VALUE The maximum string size is in the max_string_size field and the value is in the string_value field. CLC$REAL_VALUE The value is currently unimplemented. CLC$INTEGER_VALUE The value is in the integer_value field. CLC$BOOLEAN_VALUE The value is in the boolean_value field. CLC$STATUS_VALUE The value is in the status_value field.
max_string_size	Maximum string size (OST$STRING_SIZE, 0, to OSC$MAX_STRING_SIZE, 256). When writing a string variable, this field should be initialized to the same value specified when the variable was created.
string_value	Pointer to an array of one or more strings (^ array [1..*] of cell).
integer_value	Pointer to an array of one or more integers (^ array [1..*] of CLT$INTEGER, see the int field in table 9-1).
boolean_value	Pointer to an array of one or more boolean values (^ array [1..*] of CLT$BOOLEAN, see the bool field in table 9-1).
status_value	Pointer to an array of one or more status records (^ array [1..*] of CLT$STATUS). A status record is returned as a type CLT$STATUS record instead of a type OST$STATUS record so that each field can be directly referenced as if it was an SCL variable. The content of the CLT$STATUS record is the same as that of an OST$STATUS record.

(Continued)

```
variable_value.kind := clc$string_value;
variable_value.max_string_size :=
  osc$max_string_size;
variable_value.string_value := y;
```

However, if the variable is an array of strings, you can declare and initialize the value using the following statements.

```
CONST
  string_array_elements = 2;

VAR
  variable_value: clt$variable_value,
  x: record
    case 1..2 of
    =1=
{ Each array entry is the sum of the size field }
{ length (2) plus the maximum string length (256) }
      cv: array[1..
        (string_array_elements *
        (2+256))]
        of cell,
    =2=
      sv: array[1..
        string_array_elements] of
        record
          size: ost$string_size,
          value: string(256),
        recend,
    casend,
  recend,
   x.sv[1].size := 25;
   x.sv[1].value :=
     ' This is the first string';
   x.sv[2].size := 26;
   x.sv[2].value :=
     ' This is the second string';
   variable_value.kind := clc$string_value;
   variable_value.max_string_size :=
     osc$max_string_size;
   variable_value.string_value := ^x.cv;
```

CLP$CONVERT_INTEGER_TO_STRING

Purpose Converts an integer to its string representation in the specified radix.

Format **CLP$CONVERT_INTEGER_TO_STRING (int, radix, include_radix_specifier, str, status)**

Parameters **int**: integer;

Integer value.

radix: 2 .. 16;

Representation radix (2 through 16).

include_radix_specifier: boolean;

Indicates whether a trailing radix enclosed in parentheses is included.

TRUE

Radix included.

FALSE

Radix omitted.

str: VAR of ost$string;

String record.

Field	Content
size	Actual string length (0 through 256).
value	String representation (256 characters). The string data is left-justified in the 256-character field.

status: VAR of ost$status;

Status record.

CLP$CONVERT_INTEGER_TO_RJSTRING

Purpose

Converts an integer to its right-justified string representation in the specified radix.

Format

CLP$CONVERT_INTEGER_TO_RJSTRING (int, radix, include_radix_specifier, fill_character, str, status)

Parameters

int: integer;

Integer value.

radix: 2 .. 16;

Representation radix (2 through 16).

include_radix_specifier: boolean;

Indicates whether a trailing radix enclosed in parentheses is included.

TRUE

Radix included.

FALSE

Radix omitted.

fill_character: char;

Character used to fill in the string.

str: VAR of string (*);

String generated. The string length is chosen when the string variable is allocated.

status: VAR of ost$status;

Status record.

CLP$CONVERT_STRING_TO_INTEGER

Purpose Converts the string representation of an integer to the integer value.

Format **CLP$CONVERT_STRING_TO_INTEGER (str, int, status)**

Parameters **str**: string (*);

String.

int: VAR of clt$integer;

Record returned describing the integer value.

Field	Content
value	Integer value (type integer).
radix	Representation radix (2 through 16).
radix_specified	Indicates whether a radix was specified in the string. TRUE Radix specified. FALSE Radix omitted.

status: VAR of ost$status;

Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170199.

Remarks The string representation can include a leading sign and a trailing radix enclosed in parentheses.

CLP$CONVERT_STRING_TO_FILE

Purpose

Interprets a string as a file reference. It performs the following operations.

- Interprets the file reference in the string and assigns a local file name to the file.
- Establishes the validation ring of the file as the ring of the caller.

Format

CLP$CONVERT_STRING_TO_FILE (str, file, status)

Parameters

str: string (*);

String containing a file reference.

file: VAR of clt$file;

File record returned. The record consists of the following field.

local_file_name

Assigned local file name (type AMT$LOCAL_FILE_NAME).

status: VAR of ost$status;

Status record.

Condition Identifiers

Any command language condition whose code is within the range 170100 through 170199 or 170500 through 170599.

Remarks

If the record describes an integer, name, file, boolean, or status record, the procedure returns the string equivalent of the value. For a file, the full file reference is returned (beginning with a : character as described in the SCL System Interface manual). The string returned for a status record depends on the current job message level (described in chapter 6).

If the record describes an array reference, the procedure returns the following string containing the array name.

```
ARRAY: name
```

If the record describes an application value, the contents of the descriptor field of the record is returned. However, if the descriptor field is blank, the procedure returns the following string.

```
APPLICATION VALUE
```

If the record describes a value of unknown type, the procedure returns the following string.

```
UNKNOWN VALUE
```

Command Language Processing 9

Command Language Processing 9

NOS/VE allows you to define new commands. These user-defined commands are interpreted the same way system-defined SCL commands are interpreted. The commands use standard SCL command and parameter syntax.

To write a program that defines a command, you should first understand how the system processes SCL commands. Each command is processed by a part of the system called the SCL interpreter. The SCL interpreter expects the command and parameter syntax described in the SCL Language Definition manual.

At the SCL command level, the SCL interpreter recognizes only commands that are in the command list for the job. For the SCL interpreter to recognize a command you define at the SCL command level, you must add your command to the beginning of the command list for the job using the SCL command SET_COMMAND_LIST. The process of adding to the command list is described in the SCL Language Definition manual.

You add either an object library or a catalog to a command list by using SET_COMMAND_LIST. The object library or catalog must contain programs or procedures in executable form. Each program or procedure processes a command and is, therefore, referred to as a command processor.

Command Processor

The SCL interpreter accepts commands that consist of a command verb and a parameter list. (The parameter list can be empty.) When the SCL interpreter reads a command, it finds and calls the appropriate command processor, passing it the parameter list and a status variable. The command processor uses the parameter list as input information and the status variable to return its completion status. The required procedure declaration is as follows (type CLT$COMMAND).

PROCEDURE name (parameter_list: clt$parameter_list;
VAR status: ost$status);

To use its parameter list information, the command processor calls the CLP$SCAN_PARAMETER_LIST procedure to parse the parameter list according to the SCL parameter syntax rules. To parse a parameter list, the CLP$SCAN_PARAMETER_LIST command requires the parameter list that was passed to the command processor and the Parameter Descriptor Table (PDT) that defines the valid parameters for the parameter list.

You can specify the parameter definitions one per line or more than one on a line if separated by semicolons (;). The following general formats are both valid:

PDT pdt_variable_name (
parameter_definition
parameter_definition)
PDT pdt_variable_name (
parameter_definition; parameter_definition)

If a parameter definition does not fit on one line, you can use continuation lines. (A continuation line ends with an ellipsis [..].)

Each parameter definition has the following general format:

parameter_names: value_specification=default_specification

A parameter definition within a PDT declaration uses the same syntax as a parameter definition within an SCL procedure header. For a full description of the parameter definition syntax, see the SCL Language Definition manual.

The following example shows the PDT declaration that could generate a PDT for the SCL command ATTACH_FILE.

```
PDT attach_command_pdt (
  file, f : FILE = $REQUIRED
  local_file_name, lfn : NAME
  password, pw : NAME OR KEY none = none
  access_modes, access_mode, am : LIST OF KEY read,..
    append, modify, execute, shorten, write, all =
    (read, execute)
  share_modes, share_mode, sm : LIST OF KEY read,..
    append, modify, execute, shorten, write, all,..
    none = (read, execute)
  wait, w : BOOLEAN = false
  STATUS)
```

Notice that the STATUS parameter definition is a special case. It does not require a value or default specification. If the parameter name is STATUS, NOS/VE assumes that the parameter is the status variable and that it has no default.

The following defines the required parameter list for a scanner program.

```
(value_name: clt$application_value_name;
keywords: ^array[1 .. *] of ost$name;
text: string(*);
VAR value: clt$value;
VAR status: ost$status);
```

value_name

Application value name as specified in the parameter definition.

keywords

Pointer to the array of keywords defined as valid parameter values.

text

Parameter string passed to the procedure for evaluation.

value

Result of the evaluation. The parameter value must be returned as a type CLT$VALUE record (see table 9-1).

status

Status record.

Table 9-1. Evaluated Expression Value (Type CLT$VALUE)
(Continued)

Field	Content
application	Value recognized by the application (CLT$APPLICATION_VALUE, 256-character sequence). This field is generated only if the kind field is set to CLC$APPLICATION_VALUE.
var_ref	Command variable reference (CLT$VARIABLE_REFERENCE, see table 9-2). This field is generated only if the kind field is set to CLC$VARIABLE_REFERENCE.
str	String record (OST$STRING). This field is generated only if the kind field is set to CLC$STRING_VALUE. **Field** — **Content** size — Actual string length (OST$STRING_SIZE, 0 through OSC$MAX_STRING_SIZE). value — String (256 characters).
file	File record (CLT$FILE). This field is generated only if the kind field is set to CLC$FILE_VALUE. The file record consists of the following field. local_file_name Local file name (AMT$LOCAL_FILE_NAME).
name	Name record (CLT$NAME). This field is generated only if the kind field is set to CLC$NAME_VALUE. **Field** — **Content** size — Actual name length (1 through OST$MAX_NAME_SIZE). value — Name (31 characters).

(Continued)

Table 9-2. Variable Reference (CLT$VARIABLE_REFERENCE)

<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>reference</td><td>Variable reference string record (OST$STRING).
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>size</td><td>Actual string length (OST$STRING_SIZE, 0 through OSC$MAX_STRING_SIZE).</td></tr>
<tr><td>value</td><td>String (256 characters).</td></tr>
</table></td></tr>
<tr><td>lower_bound</td><td>Lower array bound (CLT$VARIABLE_DIMENSION).</td></tr>
<tr><td>upper_bound</td><td>Upper array bound (CLT$VARIABLE_DIMENSION).</td></tr>
<tr><td>value</td><td>Variable value or values (CLT$VARIABLE_VALUE).
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>descriptor</td><td>Name of the value kind (string of length OSC$MAX_NAME_SIZE, 31 characters).</td></tr>
<tr><td>kind</td><td>Key field identifying the value kind (CLT$VARIABLE_KINDS).

CLC$STRING_VALUE
The maximum string size is in the max_string_size field and the value is in the string_value field.

CLC$REAL_VALUE
The value is currently unimplemented.

CLC$INTEGER_VALUE
The value is in the integer_value field.

CLC$BOOLEAN_VALUE
The value is in the boolean_value field.

CLC$STATUS_VALUE
The value is in the status_value field.</td></tr>
</table></td></tr>
</table>

(Continued)

Retrieving Parameter List Information

Guided by the PDT, the CLP$SCAN_PARAMETER_LIST procedure parses a parameter list according to SCL syntax rules. The command processor can then use the following calls to get information about the components of the parameter list.

CLP$TEST_PARAMETER

Whether a parameter value is specified in the actual parameter list or is provided by a default value.

CLP$GET_SET_COUNT

Number of value sets specified for a parameter.

CLP$GET_VALUE_COUNT

Number of values in a value set.

CLP$TEST_RANGE

Whether the value is specified as a range.

CLP$GET_VALUE

An actual parameter value.

CLP$GET_PARAMETER

The entire parameter string.

CLP$GET_PARAMETER_LIST

The entire parameter list string.

CLP$TEST_PARAMETER

Purpose Tests whether a parameter list contains a value for the specified parameter.

Format **CLP$TEST_PARAMETER (parameter_name, parameter_specified, status)**

Parameters **parameter_name**: string (*);

Parameter name.

parameter_specified: VAR of boolean;

Indicates whether the parameter is specified.

TRUE

Parameter is specified.

FALSE

Parameter is omitted.

status: VAR of ost$status;

Status record.

Condition Identifiers

cle$unexpected_call_to
cle$unknown_parameter_name

Remarks The parameter list used is the parameter list scanned by a prior CLP$SCAN_PARAMETER_LIST call.

CLP$GET_VALUE_COUNT

Purpose Returns number of values specified in a value set.

Format **CLP$GET_VALUE_COUNT (parameter_name, value_set_number, value_count, status)**

Parameters

parameter_name: string (*);
Parameter name.

value_set_number: 1 .. clc$max_value_sets;
Value set number.

value_count: VAR of 0 .. clc$max_values_per_set;
Number of values.

status: VAR of ost$status;
Status record.

Condition Identifiers

cle$unexpected_call_to
cle$unknown_parameter_name

Remarks

- A value set is a set of values enclosed in parentheses specified for a parameter.
- The parameter list used is the parameter list scanned by a prior CLP$SCAN_PARAMETER_LIST call.

CLP$GET_VALUE

Purpose Returns a parameter value.

Format **CLP$GET_VALUE (parameter_name, value_set_number, value_number, low_or_high, value, status)**

Parameters

parameter_name: string (*);

Parameter name.

value_set_number: 1 .. clc$max_value_sets;

Value set number indicating which value set of the parameter_name is being referenced. (The number of value sets for the parameter_name is returned using the CLP$GET_SET_COUNT procedure.)

value_number: 1 .. clc$max_values_per_set;

Value number indicating which value of the value_set_number is being referenced. (The number of values in the value_set_parameter is returned using the CLP$GET_VALUE_COUNT procedure.)

low_or_high: clt$low_or_high;

Indicates whether the upper or lower bound of the range is returned.

CLC$LOW

Return the lower bound.

CLC$HIGH

Return the upper bound.

value: VAR of clt$value;

Parameter value (see table 9-1).

status: VAR of ost$status;

Status record.

Condition Identifiers

cle$unexpected_call_to
cle$unknown_parameter_name

Remarks

- The parameter list used is the parameter list scanned by a prior CLP$SCAN_PARAMETER_LIST call.
- If the parameter list of the command did not specify a value for the parameter specified on the CLP$GET_VALUE call, CLP$GET_VALUE returns value kind CLC$UNKNOWN_VALUE in the value record returned.

CLP$GET_PARAMETER_LIST

Purpose Returns the entire parameter list.

Format **CLP$GET_PARAMETER_LIST (parameter_list, status)**

Parameters **parameter_list:** VAR of ost$string;

Parameter list record.

Field	Content
size	Actual value list length (OST$STRING_SIZE, 0, through OSC$MAX_STRING_SIZE, 256).
value	Value list string (256 characters).

status: VAR of ost$status;

Status record.

Condition Identifier cle$unexpected_call_to

Remarks The parameter list used is the parameter list scanned by a prior CLP$SCAN_PARAMETER_LIST call.

CLP$GET_PATH_DESCRIPTION

Purpose Returns description of a command language file reference.

Format **CLP$GET_PATH_DESCRIPTION (file, file_reference, path_container, path, cycle_selector, open_position, status)**

Parameters **file**: clt$file;

File record consisting of the following field. This is the record returned by a CLP$GET_VALUE call for the value kind FILE.

local_file_name

File name (type AMT$LOCAL_FILE_NAME).

file_reference: VAR of clt$file_reference;

File reference record. Following are the fields and their contents.

path_name

Absolute path name (CLT$PATH_NAME, 256-character string).

path_name_size

Actual length of the path name (1 through the value of CLC$MAX_PATH_NAME_SIZE, 256).

validation_ring

Indicates whether the ring is known and if so, provides the ring number (see table 9-3).

Table 9-3. Validation Ring Specification

Field	Content
known	Key field indicating whether the validation ring is known (boolean). TRUE The validation ring is specified in the number field. FALSE The validation ring is unknown.
number	Validation ring number if known (OST$VALID_RING, 1 .. 15).

Table 9-4. Command Language Cycle Specification (CLT$CYCLE_SELECTOR)

Field	Content
specification	Indicates how the cycle is specified (CLT$CYCLE_ SPECIFICATION). CLC$CYCLE_OMITTED No cycle specified. CLC$CYCLE_SPECIFIED A cycle number was specified. CLC$CYCLE_NEXT_HIGHEST The next highest cycle was requested. CLC$CYCLE_NEXT_LOWEST The next lowest cycle was requested.
value	Actual cycle value record (PFT$CYCLE_SELECTOR).

Field	Content
cycle_option	Key field (PFT$CYCLE_OPTIONS). PFC$LOWEST_CYCLE. Lowest cycle. PFC$HIGHEST_CYCLE Highest cycle. PFC$SPECIFIC_CYCLE Cycle specified in the cycle_number field.
cycle_number	Specific cycle number (PFC$MINIMUM_ CYCLE_NUMBER through PFC$MAXIMUM_CYCLE NUMBER).

CLP$SET_WORKING_CATALOG

Purpose Sets the working catalog.

Format **CLP$SET_WORKING_CATALOG (catalog, status)**

Parameters **catalog**: string (*);

Catalog name.

status: VAR of ost$status;

Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170199 or 170500 through 170599.

Remarks The working catalog is the default catalog used if no catalog is specified in a file reference. The initial working catalog in a job is the $LOCAL catalog. You can change the working catalog with a CLP$SET_WORKING_CATALOG call or the SCL command SET_WORKING_CATALOG.

CLP$PUSH_PARAMETERS

Purpose Establishes the environment for scanning a parameter list.

Format **CLP$PUSH_PARAMETERS (status)**

Parameter **status**: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks After scanning the parameter list and retrieving parameter values, the program calls CLP$POP_PARAMETERS to return to the previous environment.

Command Utility

Each task has its own SCL command stack. The first entry on the stack is the current SCL command list for the job. A task can push and pop command list entries from its stack.

A command utility is a task that adds its own entry to its command stack so that it can process subcommands. To do so, it performs the following steps.

1. Defines its subcommand list and function list. The list specifies a command processor for each subcommand and function.

2. Calls CLP$PUSH_UTILITY to establish the utility command environment. CLP$PUSH_UTILITY pushes the subcommand list and function list on the task's SCL command stack and allocates storage for utility command variables.

3. Calls CLP$SCAN_COMMAND_FILE to call the SCL interpreter to process command input. The SCL interpreter processes each command entry. If the command entered is a utility subcommand, the SCL interpreter calls the command processor specified in the utility command list.

4. Calls CLP$END_SCAN_COMMAND_FILE to direct the SCL interpreter to stop processing command input for the utility. It is normally called from the utility subcommand processor that terminates utility processing (such as the QUIT processor in the command utility example in this section).

5. Calls CLP$POP_UTILITY to disestablish the utility environment. It removes the utility command and function list from the SCL command stack.

Writing a program as a command utility has the following advantages.

- The utility writer does not write routines to parse commands or parameter lists or call the appropriate command processors.

- Utility users can enter SCL statements controlling the order of command execution (such as iteration and condition checks) within the subcommand sequence.

- The command syntax for the utility is the SCL command syntax with which the utility user is already familiar.

CLP$PUSH_UTILITY

Purpose Establishes a new command environment.

Format **CLP$PUSH_UTILITY (utility_name, search_mode, commands, functions, status)**

Parameters **utility_name**: ost$name;

Command environment name.

search_mode: clt$command_search_modes;

Command list search mode.

CLC$GLOBAL_COMMAND_SEARCH

All command lists searched; escape mode allowed.

CLC$RESTRICTED_COMMAND_SEARCH

Except in escape mode, only the utility command list is searched; in escape mode, all command lists except the utility command list are searched.

CLC$EXCLUSIVE_COMMAND_SEARCH

Only the utility command list is searched; escape mode is not allowed.

functions: ^clt$function_list;

Utility function list pointer. The list is an adaptable array of one or more CLT$FUNCTION_LIST_ENTRY records. Each record has the following fields.

name

Function name (OST$NAME, 31 characters).

kind

Key field (CLT$FUNCTION_LIST_ENTRY_KIND).

CLC$LINKED_FUNCTION

The function processor is already loaded and linked in the task's address space. The function pointer is in the function field.

CLC$UNLINKED FUNCTION

The function processor is not yet loaded or linked. The function module name is in the procedure_name field.

function

Pointer to the function procedure (CLT$FUNCTION). This field is generated only if the kind field is CLC$LINKED_ FUNCTION.

procedure_name

Name of the function procedure that must be loaded before it is called (PMT$PROGRAM_NAME). This field is generated only if the kind field is CLC$UNLINKED_ FUNCTION.

status: VAR of ost$status;

Status record.

Condition Identifier

None.

CLP$POP_UTILITY

Purpose	Disestablishes the most recently established command environment.
Format	**CLP$POP_UTILITY (status)**
Parameter	**status**: VAR of ost$status; Status record.
Condition Identifier	cle$unexpected_call_to

CLP$SCAN_COMMAND_FILE

Purpose
Calls the SCL interpreter to read and interpret command input from the specified file.

Format
CLP$SCAN_COMMAND_FILE (file, utility_name, prompt_string, status)

Parameters
file: amt$local_file_name;

Local file name. Usually, the file is the current command input file (referenced as CLC$CURRENT_COMMAND_INPUT).

utility_name: ost$name;

Name of the utility that uses the command input as specified on a previous CLP$PUSH_UTILITY call.

prompt_string: string (*);

Prompt string used if the command file is assigned to an interactive terminal.

status: VAR of ost$status;

Status record.

Condition Identifiers
All command language conditions.

Remarks

- The SCL interpreter processes the commands on the specified file as if the commands were a statement list of an unlabeled block statement.

- To end command interpretation prior to reaching the end-of-information on the command file, the task must call the CLP$END_SCAN_COMMAND_FILE procedure. The CLP$END_SCAN_COMMAND_FILE call is usually issued within the command processor that ends utility processing.

Assuming the object module generated by compilation of the program is on file LGO, the following statement sequence shows how to generate an object library containing the module, add the object library to the command list, and then execute the utility.

```
/create_object_library
COL/add_module library=lgo
COL/generate_library library=my_library
COL/quit
/set_command_list add=my_library
/info_please
What information do you want?
Enter an information command or
  enter quit to leave the utility.
To display:
  --processor attributes, enter:  processor
  --SRUs accumulated for the job, enter:  srus
  --CPU time accumulated for the task, enter:  cp_time
  --account and project numbers:  acct_proj
Info item?/@rpcesspr
--ERROR-- @RPCESSPR is not a command.
Info item?/processor
Processor attributes:
  CPU model P3
  Serial number 2
  Page size 8192 bytes
Info item?/srus
0499116 SRUs.
Info item?/cp_time
Accumulated CPU time for the task
  434051 microseconds in job mode
  9  microseconds in monitor mode
Info item?/acct_proj
Account D5923
Project P693N354
Info item?/quit
Bye now.
/
```

```
{ This procedure writes a message to the output file. }
{ It assumes that the output file has been opened and }
{ its file identifier returned in a variable named }
{ output_fid. }

PROCEDURE put_message (message: string( * ));

  VAR
    byte_address: amt$file_byte_address,
    stat: ost$status;

amp$put_next (output_fid, #LOC(message),
  #SIZE(message), byte_address, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;
PROCEND put_message;

{ This procedure displays instructions after a user }
{ enters the info_please command. }

PROCEDURE display_instructions;

put_message (' What information do you want?');
put_message (' Enter an information command or ');
put_message ('   enter quit to leave the utility.');
put_message (' To display:');
put_message
  ('   --processor attributes, enter:  processor');
put_message
  ('   --SRUs accumulated for the job, enter:  srus');
put_message
  ('   --CPU time accumulated for the task, enter:  cp_time');
put_message
  ('   --account and project numbers:  acct_proj');

PROCEND display_instructions;
```

```
PUSH message_ptr_1: [17+serial_no_string.size];
message_ptr_1^(1,17) := '   Serial number ';
message_ptr_1^(18,serial_no_string.size) :=
  serial_no_string.value(1,serial_no_string.size);
put_message(message_ptr_1^);

clp$convert_integer_to_string (attributes.page_size,
  10, false, page_size_string, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

PUSH message_ptr_2: [19+page_size_string.size];
message_ptr_2^(1,13) := '   Page size ';
message_ptr_2^(14,page_size_string.size) :=
  page_size_string.value(1,page_size_string.size);
message_ptr_2^((page_size_string.size+14),6) :=
  ' bytes';
put_message (message_ptr_2^);

PROCEND processor_command;
```

```
{ This procedure processes the cp_time command. }
{ It returns the number of microseconds accumulated in }
{ job mode and in monitor mode for the task. }

PROCEDURE cp_time_command(cp_time_parameter_list:
  clt$parameter_list;
  VAR stat: ost$status);

  VAR
    cp_time: pmt$task_cp_time,
    job_mode_string: ost$string,
    monitor_mode_string: ost$string,
    message_ptr_1, message_ptr_2: ^string(*);

pmp$get_task_cp_time(cp_time, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

put_message(' Accumulated CPU time for the task');

clp$convert_integer_to_string(cp_time.task_time, 10,
  false, job_mode_string, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;
```

```
{ This procedure processes the acct_proj command. }
{ It returns the account and project names for the }
{ job. }

PROCEDURE acct_proj_command
  (acct_proj_parameter_list: clt$parameter_list;
  VAR stat: ost$status);

  VAR
    account: avt$account_name,
    project: avt$project_name,
    message1, message2: string(osc$max_name_size+9);

pmp$get_account_project(account,project, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

message1(1,9) := ' Account ';
message1(10,STRLENGTH(account)) := account;
put_message (message1);

message2(1,9) := ' Project ';
message2(10,STRLENGTH(project)) := project;
put_message(message2);

PROCEND acct_proj_command;

{ This procedure processes the quit command.  It }
{ sends a message and then ends the command file }
{ scan by the SCL interpreter. }

PROCEDURE quit_command (quit_parameter_list:
  clt$parameter_list; VAR stat: ost$status);

put_message (' Bye now.');

clp$end_scan_command_file(utility_name, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;
PROCEND quit_command;
```

```
{ STATUS }
    [[clc$optional],
    1, 1,
    1, 1,
    clc$value_range_not_allowed,
    [NIL,
    clc$variable_reference, clc$array_not_allowed,
      clc$status_value]]];

VAR
  info_please_pdt_dv1: [STATIC, READ,
    cls$pdt_names_and_defaults] string(7) := '$output';

?? POP ??

clp$scan_parameter_list (parameter_list,
  info_please_pdt, status);
IF NOT status.normal THEN
  pmp$exit(status);
IFEND;

{ The following calls get the input and output file }
{ names and open the input and output files. }

clp$get_value('output',1,1,clc$low,output_file,status);
IF NOT status.normal THEN
  RETURN;
IFEND;

amp$open(output_file.file.local_file_name, amc$record,
  NIL, output_fid, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

display_instructions;
```

```
{ The following call directs the SCL interpreter to }
{ begin interpreting the commands entered in the }
{ input file.  It prompts for command input with the }
{ string Info item?  When it reads a command, it }
{ finds the command processor, calls it, and passes }
{ the parameter list to it. }

clp$scan_command_file (clc$current_command_input,
  utility_name, 'Info item?', status);
IF NOT status.normal THEN
  RETURN;
IFEND;

amp$close(output_fid, status);

clp$pop_utility(status);

PROCEND info_please;
MODEND command_utility_example;
```

Table 9-5. Argument Descriptor (CLT$ARGUMENT_DESCRIPTOR)

<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>required_or_
optional</td><td>Indicates whether the parameter is required or optional and its default, if any (CLT$REQUIRED_OR_OPTIONAL).
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>selector</td><td>Key field determining whether the parameter is required or optional.

CLC$REQUIRED
The parameter is required; no default value is supplied.

CLC$OPTIONAL
The parameter is optional; no default value is supplied.

CLC$OPTIONAL_WITH_DEFAULT
The parameter is optional; the default field is generated to supply the default value.</td></tr>
<tr><td>default</td><td>Pointer to the default value (^ string(*)).</td></tr>
</table>
</td></tr>
<tr><td>value_kind_
specifier</td><td>Value kind specifier (CLT$VALUE_KIND_SPECIFIER, see table 9-6).</td></tr>
</table>

Table 9-6. Value Kind Specifier (CLT$VALUE_KIND_SPECIFIER) *(Continued)*

Field	Content
	CLC$REAL_VALUE This value is currently unimplemented. CLC$BOOLEAN_VALUE Boolean value. CLC$STATUS_VALUE Status record.
array_ allowed	Indicates whether the command variable can be an array (used only if kind is CLC$VARIABLE_ REFERENCE). CLC$ARRAY_NOT_ALLOWED The variable cannot be an array. CLC$ARRAY_ALLOWED The variable can be an array.
variable_ kind	Indicates the variable type or the type of each element in the array variable (used only if kind is CLC$VARIABLE_REFERENCE). CLC$STRING_VALUE String. CLC$REAL_VALUE This value is currently unimplemented. CLC$INTEGER_VALUE Integer. CLC$BOOLEAN_VALUE Boolean. CLC$STATUS_VALUE Status record. CLC$ANY_VALUE Any type.
value_name	Name passed to the application value scanner (CLT$APPLICATION_VALUE_NAME). It is used only if the kind is CLC$APPLICATION_VALUE).

(Continued)

CLP$SCAN_ARGUMENT_LIST

Purpose Scans the argument list of a function.

Format **CLP$SCAN_ARGUMENT_LIST (function_name, argument_list, adt, avt, status)**

Parameters

function_name: clt$name;

Function name record.

Field	Content
size	Actual name length (OST$NAME_SIZE, 1 through OSC$MAX_NAME_SIZE).
value	Function name string (OST$NAME, 31 characters).

argument_list: string (*);

Argument list.

adt: ^ clt$argument_descriptor_table;

Pointer to the argument descriptor table.

avt: ^ clt$argument_value_table;

Pointer to the argument value table.

status: VAR of ost$status;

Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170599 or 171000 through 171099.

Table 9-7. Token Record (CLT$TOKEN) *(Continued)*

<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>str</td><td>String record (OST$STRING). This field is generated only if the kind field value is CLC$STRING_TOKEN.
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>size</td><td>Actual string length (OST$STRING_SIZE, 0 through OSC$MAX_STRING_SIZE).</td></tr>
<tr><td>value</td><td>String (256 characters).</td></tr>
</table></td></tr>
<tr><td>name</td><td>Name (CLT$NAME). This field is generated only if the kind field value is CLC$NAME_TOKEN.
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>size</td><td>Actual name length (1 through OSC$MAX_NAME_SIZE).</td></tr>
<tr><td>value</td><td>Name (31 characters).</td></tr>
</table></td></tr>
<tr><td>int</td><td>Integer value (CLT$INTEGER). This field is generated only if the kind field value is CLC$INTEGER_TOKEN.
<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>value</td><td>Integer value (integer).</td></tr>
<tr><td>radix</td><td>Radix used (2 through 16).</td></tr>
<tr><td>radix_specified</td><td>Indicates whether a radix was specified.
TRUE
Radix specified.
FALSE
Radix omitted.</td></tr>
</table></td></tr>
<tr><td>rnum</td><td>Floating point value (CLT$REAL). (Although CLP$SCAN_TOKEN recognizes real number input, real number processing is currently unimplemented.)</td></tr>
</table>

CLP$SCAN_TOKEN

Purpose Scans the next lexical unit.

Format **CLP$SCAN_TOKEN (text, index, token, status).**

Parameters

text: string (*);
Text to be scanned.

index: VAR of ost$string_index; [input, output]
Index to next character (input and output value).

token: VAR of clt$token;
Lexical unit.

status: VAR of ost$status;
Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170199.

CLP$SCAN_EXPRESSION

Purpose Scans and evaluates an expression.

Format **CLP$SCAN_EXPRESSION (expression, value_kind_specifier, value, status)**

Parameters **expression**: string (*);

Expression.

value_kind_specifier: clt$value_kind_specifier;

Value kind specifier.

value: VAR of clt$value;

Value of expression.

status: VAR of ost$status;

Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170599 or 171000 through 171099.

CLP$COLLECT_COMMANDS

Purpose Collects commands on the specified file.

Format **CLP$COLLECT_COMMANDS (local_file_name, terminator, status)**

Parameters **local_file_name**: amt$local_file_name;

Local file name of the file on which the commands are collected.

terminator: ost$name;

Name that terminates the copy.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks

- CLP$COLLECT_COMMANDS is designed for use by a utility subcommand processor that requires input from the command file.
- CLP$COLLECT_COMMANDS directs the SCL interpreter to copy commands read from the command file to another file without interpreting or processing the commands. It continues copying commands until it reads the specified terminator (the terminator is not copied).
- CLP$COLLECT_COMMANDS writes one command per line on the specified file even if more than one command was entered on a line of the command file. For example, suppose the specified terminator name is BREAKEND and CLP$COLLECT_COMMANDS reads the following line from the command file.

  ```
  disl o=my_file;copf listing;breakend
  ```

  CLP$COLLECT_COMMANDS writes the following output on the specified file.

  ```
  disl o=my_file
  copf listing
  ```

CLP$GET_DATA_LINE

Purpose Reads the next line from the current command file.

Format **CLP$GET_DATA_LINE (prompt_string, line, got_line, status)**

Parameters **prompt_string**: string (*);

Prompt string used if the command file is assigned to an interactive terminal.

line: VAR of ost$string;

Line read (up to 256 characters).

got_line: VAR of boolean;

Indicates whether a line was read or the end-of-information encountered.

TRUE

A line was read.

FALSE

No line was read; the call read the end-of-information indicator.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks

- CLP$GET_DATA_LINE directs the SCL interpreter to return the next line read from the command file without interpreting or processing the line.
- CLP$GET_DATA_LINE is designed for use by a utility subcommand processor that requires input from the command file.

Scanning Declarations

You can call CLP$SCAN_PROC_DECLARATION to parse a PDT declaration. The SCL interpreter calls CLP$SCAN_PROC_DECLARATION to parse an SCL procedure header. The INPUT_TYPE parameter on the CLP$SCAN_PROC_DECLARATION call specifies the type of input provided.

CLP$SCAN_PROC_DECLARATION requires you to provide a procedure to preprocess its input. The procedure pointer must be of type CLT$PROC_INPUT_PROCEDURE. The following is the CLT$PROC_INPUT_PROCEDURE type declaration.

```
clt$proc_input_procedure = ^procedure
  (VAR line: ost$string;
  VAR index: ost$string_index;
  VAR token: clt$token;
  VAR status: ost$status);
```

The first parameter returns the next input line. The second parameter returns the position of the character following the first token in the line. The third parameter returns the first token in the line.

The preprocessing procedure must perform the following tasks.

- Determine whether a line is a continuation line. If it is, the procedure must remove the ellipsis (..) and concatenate the line with the next line.
- Discard all lines that contain only spaces or comments.
- Return an empty line (a line of size zero) when it reads the end of the input.

CLP$SCAN_PROC_DECLARATION

Purpose Parses a PDT declaration or an SCL procedure header.

Format **CLP$SCAN_PROC_DECLARATION (input_type, get_line, proc_name_area, parameter_name_area, parameter_area, symbolic_parameter_area, extra_info_area, proc_names, pdt, symbolic_parameters, status)**

Parameters

input_type: clt$proc_input_type;

Indicates whether the input is a PDT declaration or an SCL procedure header.

CLC$PROC_INPUT

The input is an SCL procedure header.

CLC$PDT_INPUT

The input is a PDT declaration.

get_line: clt$proc_input_procedure;

Pointer to the procedure that preprocesses the input.

proc_name_area: VAR of SEQ (*);

Adaptable sequence in which the procedure CLP$SCAN_PROC_DECLARATION stores the procedure names as an array.

parameter_name_area: VAR of SEQ (*);

Adaptable sequence in which the procedure CLP$SCAN_PROC_DECLARATION stores the parameter names as an array.

parameter_area: VAR of SEQ (*);

Adaptable sequence in which the procedure CLP$SCAN_PROC_DECLARATION stores the parameter descriptors.

symbolic_parameter_area: VAR of SEQ (*);

Adaptable sequence in which the procedure CLP$SCAN_PROC_DECLARATION stores the original unevaluated form of any expressions. (When generating CYBIL statements, GENPDT uses the original expression within the statement.)

PDT Pointers

A PDT (as described in table 9-9) contains two pointers: a pointer to a list of possible parameter keywords and a pointer to a list of parameter descriptors. Each entry in the list of parameter keywords references an entry in the list of parameter descriptors.

Table 9-9. Parameter Descriptor Table (Type CLT$PARAMETER_DESCRIPTOR_TABLE)

<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>names</td><td>Pointer to an array listing all parameter names (^array [1..*] of CLT$PARAMETER_NAME_DESCRIPTOR).

The order of the names in the array is irrelevant except when an error in a positional parameter is reported. The error is reported using the name in that position of the names array.

<table>
<tr><th>Field</th><th>Content</th></tr>
<tr><td>name</td><td>Name used to specify the parameter when it is specified by name, rather than position (type OST$NAME).</td></tr>
<tr><td>number</td><td>Index into the parameter descriptors array of the entry describing the parameter (1 through CLC$MAX_PARAMETERS).

More than one name can reference the same parameter descriptor.</td></tr>
</table></td></tr>
<tr><td>parameters</td><td>Pointer to an array listing the parameter descriptors (^ array [1 .. *] of CLT$PARAMETER_DESCRIPTOR; see table 9-10).

The order of the entries in the parameters array must correspond to the positional order of the parameters in the parameter list. One entry must exist for each parameter.</td></tr>
</table>

Parameter Descriptor

Each parameter that can be specified on a command must have a parameter descriptor that can be referenced via the command processor PDT. The parameter descriptor specifies the valid syntax of the parameter specification.

SCL allows a parameter specification to specify more than one value. It can be a series of one or more value sets with one or more values in each set. Each value can specify a single value or a range of values.

For example, the following could be a parameter specification:

((23,4,5),6,(12..15,2))

It specifies three value sets. The first value set contains three values: 23, 4, and 5. The second value set contains one value: 6. The third value set contains two values: 12..15 and 2.

A parameter descriptor as described in table 9-10 provides the following information about a parameter:

- Whether the parameter is required or optional. For an optional parameter, it indicates whether the parameter has a default value and, if it has one, the default value itself.
- The maximum and minimum number of value sets allowed for the parameter.
- The maximum and minimum number of values allowed within a value set for the parameter.
- Whether a value for the parameter can be specified as a range.
- The value kind specifier for the parameter.

Value Kind Specifier

Each parameter descriptor specifies a value kind specifier (see table 9-6). The value kind specifier describes a valid value for the parameter.

SCL processes each parameter value as an expression to be evaluated. The result of the expression evaluation is a CLT$VALUE record (described in table 9-1).

An expression can be evaluated as the value itself or as a keyword, array, or command variable reference that specifies the value.

The value kind specifier provides the following additional information, depending on the value kind:

- Name: Maximum and minimum name length.
- String: Maximum and minimum string length.
- Integer: Maximum and minimum value.
- Keyword: Pointer to the list of valid keywords.
- Command variable: Variable type and whether the variable can be an array.
- Application value: Application value scanner and the value name passed to the scanner. The application value scanner is executed to evaluate the expression.

Appendixes

Glossary A

A

Abort

The immediate abnormal termination of a task.

B

Beginning-of-Information (BOI)

The point at which file data begins. The byte address at the beginning-of-information is always zero.

C

Catalog

A directory of files and catalogs maintained by the system for a user. The catalog $LOCAL contains only file 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.

Command Utility

A NOS/VE processor that adds its command list (referred to as its subcommands) to the beginning of the SCL command list. The subcommands are removed from the command list when the processor terminates.

Condition Handler

A statement or procedure to which control is transferred when a condition occurs. The statement or procedure is executed only if it has been established as the condition handler for the specified condition and the condition occurs in its scope.

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.

L

List

A command format notation specifying that a parameter can be given several values. See Value List.

O

Object File

A file containing one or more object modules.

Object Module

A compiler-generated unit containing object code and instructions for loading the object code. It is accepted as input by the loader and the object library generator.

Q

Queue

A sequence of messages. Tasks can communicate by adding and removing messages in a queue to which the tasks are connected.

R

Record

A unit of data than can be read or written by a single I/O request.

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.

U

Utility

A NOS/VE processor consisting of routines that perform a specific operation. See Command Utility.

V

Value

An expression or application value specified in a parameter list. Each value must match the defined kind of value for the parameter. Keywords, constants, and variable references are all values.

Value Count

An integer expression indicating the number of value elements supplied for a parameter.

Value Element

A single value or a range of values represented by two values separated by an ellipsis. For example:

value or value..value

See Value, Value List, and Value Set.

Value List

A series of value sets separated by spaces or commas and enclosed in parentheses. If only one value set is given in the list, the parentheses can be omitted. For example:

value set or (value set,value set,value set)

See Value, Value Element, and Value Set.

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 left-most bit is always zero.

Table B-1. ASCII Character Set *(Continued)*

ASCII Code			Graphic or	
Decimal	Hexadecimal	Octal	Mnemonic	Name or Meaning
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)

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.

AV

Types

```
avt$account_name = ost$name;

avt$project_name = ost$name;
```

CL

Constants

```
clc$assign_token = clc$eq_token;
clc$command_language_id = 'cl';
clc$current_command_input =
  '$COMMAND                         ';
clc$echoed_commands =
  '$ECHO                            ';
clc$error_output =
  '$ERRORS                          ';
clc$job_command_input =
  'COMMAND                          ';
clc$job_command_response =
  '$RESPONSE                        ';
clc$job_input =
  'INPUT                            ';
clc$job_output =
  'OUTPUT                           ';
clc$keyword_value = clc$unknown_value;
clc$listing_output =
  '$LIST                            ';
```

Types

```
clt$application_value = SEQ (ost$string);

clt$application_value_name = ost$name;

clt$application_value_scanner =
  procedure (value_name: clt$application_value_name;
    keyword_values: ^array [1 .. * ] of ost$name;
    text: string ( * );
  VAR value: clt$value;
  VAR status: ost$status);

clt$argument_descriptor = record
  required_or_optional: clt$required_or_optional,
  value_kind_specifier: clt$value_kind_specifier,
recend;

clt$argument_descriptor_table = array [1 .. * ]
  of clt$argument_descriptor;

clt$argument_value_table = array [1 .. * ]
  of clt$value;

clt$av_scanner_kind = (clc$unspecified_av_scanner,
  clc$linked_av_scanner, clc$unlinked_av_scanner);
```

```
clt$function_list = array [1 .. * ] of
  clt$function_list_entry;

clt$function_list_entry = record
  name: ost$name,
  case kind: clt$function_list_entry_kind of
  = clc$linked_function =
    func: clt$function,
  = clc$unlinked_function =
    procedure_name: pmt$program_name,
  casend,
recend;

clt$function_list_entry_kind = (clc$linked_function,
  clc$unlinked_function);

clt$integer = record
  value: integer,
  radix: 2 .. 16,
  radix_specified: boolean,
recend;

clt$lexical_kinds = (clc$unknown_token,
  clc$space_token, clc$eol_token, clc$dot_token,
  clc$semicolon_token, clc$colon_token,
  clc$lparen_token, clc$lbracket_token,
  clc$lbrace_token, clc$rparen_token,
  clc$rbracket_token, clc$rbrace_token,
  clc$uparrow_token, clc$rslant_token,
  clc$query_token, clc$comma_token,
  clc$ellipsis_token, clc$exp_token, clc$add_token,
  clc$sub_token, clc$mult_token, clc$div_token,
  clc$cat_token, clc$gt_token, clc$ge_token,
  clc$lt_token, clc$le_token, clc$eq_token,
  clc$ne_token, clc$string_token, clc$name_token,
  clc$integer_token, clc$real_token);

clt$low_or_high = (clc$low, clc$high);

clt$name = record
  size: ost$name_size,
  value: ost$name,
recend;
```

```
clt$proc_input_type = (clc$proc_input,
  clc$pdt_input);

clt$proc_names = array [1 .. * ] of ost$name;

clt$real = record
  significant_digits: 1 ..
  clc$max_significant_digits,
  preferred_exponent: integer,
  value: array [1 .. 16] of cell,
recend;

clt$required_or_optional = record
  case selector: (clc$required, clc$optional,
    clc$optional_with_default) of
  = clc$required =
    ,
  = clc$optional =
    ,
  = clc$optional_with_default =
    default: ^string ( * ),
  casend,
recend;

clt$status = record
  normal: clt$boolean,
  identifier: clt$status_identifier,
  condition: clt$integer,
  text: ost$string,
recend;

clt$status_identifier = record
  size: ost$string_size,
  value: string (2),
recend;

clt$sub_command_list = array [1 .. * ] of
  clt$sub_command_list_entry;

clt$sub_command_list_entry = record
  name: ost$name,
  case kind: clt$sub_command_list_entry_kind of
  = clc$linked_sub_command =
    command: clt$command,
  = clc$unlinked_sub_command,
    clc$procedure_sub_command =
    procedure_name: pmt$program_name,
  casend,
recend;
```

```
  = clc$real_value =
    rnum: clt$real,
  = clc$integer_value =
    int: clt$integer,
  = clc$boolean_value =
    bool: clt$boolean,
  = clc$status_value =
    status: ost$status,
  casend,
recend;

clt$value_kind_specifier = record
  keyword_values: ^array [1 .. * ] of ost$name,
  case kind: clt$value_kinds of
  = clc$keyword_value, clc$any_value =
    ,
  = clc$variable_reference =
    array_allowed: (clc$array_not_allowed,
      clc$array_allowed),
    variable_kind: clc$string_value ..
      clc$any_value,
  = clc$application_value =
    value_name: clt$application_value_name,
    scanner: record
      case kind: clt$av_scanner_kind of
      = clc$unspecified_av_scanner =
        ,
      = clc$linked_av_scanner =
        proc: ^clt$application_value_scanner,
      = clc$unlinked_av_scanner =
        name: pmt$program_name,
      casend,
    recend,
  = clc$file_value =
    ,
  = clc$name_value =
    min_name_size: ost$name_size,
    max_name_size: ost$name_size,
  = clc$string_value =
    min_string_size: ost$string_size,
    max_string_size: ost$string_size,
  = clc$integer_value =
    min_integer_value: integer,
    max_integer_value: integer,
  = clc$real_value, clc$boolean_value,
    clc$status_value =
    ,
  casend,
recend;
```

```
{ Status variables are mapped to clt$status records }
{ rather than ost$status records so that the individual }
{ fields of an SCL status variable can be directly }
{ referenced as if they were SCL variables of the }
{ appropriate kind. The size subfields of the }
{ identifier and text fields of a cltstatus record }
{ represent the correspondiing current_string_size. }
        status_value: ^array [1 .. * ] of clt$status,
      casend,
    recend;

    clt$variable_scope = record
      case kind: clt$variable_scope_kind of
      = clc$local_variable .. clc$xref_variable =
        ,
      = clc$utility_variable =
        utility_name: ost$name,
      casend,
    recend;

    clt$variable_scope_kind = (clc$local_variable,
      clc$job_variable, clc$xdcl_variable,
      clc$xref_variable, clc$utility_variable);
```

JM

Constants

```
jmc$job_management_id = 'JM';
jmc$job_sequence_number_size = 5;
jmc$null_job_sequence_number = '    $';
jmc$sru_count_max = 0ffffffffffff(16);
jmc$time_limit_condition = 1;
```

Types

```
jmt$job_mode = (jmc$batch,jmc$interactive_connected,
  jmc$interactive_cmnd_disconnect,
  jmc$interactive_line_disconnect,
  jmc$interactive_sys_disconnect);

jmt$job_resource_condition =
  pmt$condition_identifier;

jmt$job_sequence_number =
  string (jmc$job_sequence_number_size);

jmt$queue_reference_name = ost$name;

jmt$sru_count = 0 .. jmc$sru_count_max;
```

OF

Constants

```
ofc$operator_facility_id = 'OF';
ofc$max_send_message = 64;
ofc$max_display_message = 64;
```

Types

```
oft$operator_id = ost$name;
```

Types

```
ost$activity = record
  case activity: ost$wait_activity OF
   =osc$await_time=
     milliseconds: 0 .. OFFFFFFFF(16),
   =pmc$await_task_termination=
     task_id: pmt$task_id,
   =pmc$await_local_queue_message=
     qid: pmt$queue_connection,
  casend,
recend;

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$date = record
  case date_format: ost$date_formats of
  = osc$month_date =
    month: ost$month_date, { month DD, YYYY }
  = osc$mdy_date =
    mdy: ost$mdy_date, { MM/DD/YY }
  = osc$iso_date =
    iso: ost$iso_date, { YYYY-MM-DD }
  = osc$ordinal_date =
    ordinal: ost$ordinal_date, { YYYYDDD }
  = osc$dmy_date =
    dmy: ost$dmy_date { DD/MM/YY }
  casend,
recend;
```

```
ost$minimum_save_area = packed record
  p_register: ost$p_register,
  vmid: ost$virtual_machine_identifier,
  undefined: 0 .. 0fff(16),
  a0_dynamic_space_pointer: ^cell,
  frame_descriptor: ost$frame_descriptor,
  a1_current_stack_frame: ^cell,
  user_mask: ost$user_conditions,
  a2_previous_save_area: ^ost$stack_frame_save_area,
recend;

ost$monitor_condition =
  (osc$detected_uncorrected_err, osc$not_assigned,
  osc$short_warning, osc$instruction_spec,
  osc$address_specification, osc$exchange_request,
  osc$access_violation, osc$environment_spec,
  osc$external_interrupt, osc$page_fault,
  osc$system_call, osc$system_interval_timer,
  osc$invalid_segment_ring_0,
  osc$out_call_in_return, osc$soft_error,
  osc$trap_exception);

ost$monitor_conditions = set OF
  ost$monitor_condition;

ost$month_date = string (18);

ost$name = string (osc$max_name_size);

ost$name_size = 1 .. osc$max_name_size;

ost$ordinal_date = string (7);

ost$p_register = PACKED record
  undefined1: 0 .. 3(16),
  global_key: ost$key_lock_value,
  undefined2: 0 .. 3(16),
  local_key: ost$key_lock_value,
  pva: ost$pva,
recend;

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$status_message_level =
  (osc$current_message_level,
  osc$brief_message_level, osc$full_message_level,
  osc$explain_message_level);

ost$status_message_line = string ( * );

ost$status_message_line_count = 0 ..
  osc$max_status_message_lines;

ost$status_message_line_size = 0 ..
  osc$max_status_message_line;

ost$status_severity = (osc$informative_status,
  osc$warning_status, osc$error_status,
  osc$fatal_status, osc$catastrophic_status);

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),
    second: string (2),
  casend,
recend;
```

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$permanent_file_id = 'PF';

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$charge_change =
    ,
  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$family_name,
      account: avt$account_name,
      user: ost$user_name,
    recend,
  = pfc$member =
    member_description: record
      family: ost$family_name,
      account: avt$account_name,
      project: avt$project_name,
      user: ost$user_name,
    recend,
  casend,
recend;
```

PM

Constants

```
pmc$debug_mode_on = TRUE;
pmc$debug_mode_off = FALSE;

pmc$max_connected_per_queue = 20;
pmc$max_library_list = 0ffff(16);
pmc$max_messages_per_queue = 100;
pmc$max_module_list = 0ffff(16);
pmc$max_object_file_list = 0ffff(16);
pmc$max_queues_per_job = 20;
pmc$max_segs_per_message = 12;
pmc$max_task_id = 0ffffffff(16);

pmc$maximum_pit_value = 7fffffff(16);
pmc$minimum_pit_value = 1000;
pmc$program_management_id = 'PM';
```

Types

```
pmt$ascii_logs = pmc$system_log .. pmc$job_log;

pmt$ascii_logset = set of pmt$ascii_logs;

pmt$binary_logs = pmc$job_statistic_log ..
  pmc$statistic_log;

pmt$binary_logset = set of pmt$binary_logs;

pmt$block_exit_reason = set of (pmc$block_exit;
  pmc$program_termination, pmc$program_abort);

pmt$connected_tasks_per_queue = 0 ..
  pmc$max_connected_per_queue;
```

```
pmt$condition_identifier = 0 .. 255;

pmt$condition_information = cell;

pmt$condition_name = ost$name;

pmt$condition_selector = (pmc$all_conditions,
  pmc$system_conditions, pmc$block_exit_processing,
  jmc$job_resource_condition,
  mmc$segment_access_condition,
  ifc$interactive_condition, pmc$pit_condition,
  pmc$user_defined_condition,
  pmc$condition_combination);

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$debug_mode = boolean;

pmt$established_handler = record
  established: boolean,
  est_handler_stack: ^pmt$established_handler,
  handler: pmt$condition_handler,
  established_conditions: pmt$condition,
  handler_active: pmt$condition_handler_active,
recend;

pmt$global_logs = pmc$account_log .. pmc$system_log;

pmt$global_binary_logs = pmc$account_log ..
  pmc$statistic_log;

pmt$global_binary_logset = set of
  pmt$global_binary_logs;

pmt$global_logset = set of pmt$global_logs;

pmt$initialization_value = (pmc$initialize_to_zero,
  pmc$initialize_to_alt_ones,
  pmc$initialize_to_indefinite,
  pmc$initialize_to_infinity);
```

```
pmt$message_kind = (pmc$message_value,
  pmc$no_message, pmc$passed_segments,
  pmc$shared_segments);

pmt$message_value = SEQ (REP 1 of
  pmt$segments_per_message, REP
  pmc$max_segs_per_message of pmt$queued_segment);

pmt$messages_per_queue = 0 ..
  pmc$max_messages_per_queue;

pmt$module_list = array [1 .. * ] of
  pmt$program_name;

pmt$number_of_libraries = 0 .. pmc$max_library_list;

pmt$number_of_modules = 0 .. pmc$max_module_list;

pmt$number_of_object_files = 0 ..
  pmc$max_object_file_list;

pmt$object_file_list = array [1 .. * ] of
  amt$local_file_name;

pmt$object_library_list = array [1 .. * ] of
  amt$local_file_name;

pmt$os_name = string (22); { NOS/VE Rnn Level nnnn }

pmt$pit_value = pmc$minimum_pit_value ..
  pmc$maximum_pit_value;

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$prog_description_contents = set of
  pmt$prog_description_content;

pmt$program_name = ost$name;

pmt$program_parameters = SEQ ( * );

pmt$queue_connection = 1 .. pmc$max_queues_per_job;

pmt$queue_limits = record
  maximum_queues: pmt$queues_per_job,
  maximum_connected: pmt$connected_tasks_per_queue,
  maximum_messages: pmt$messages_per_queue,
recend;

pmt$queue_name = ost$name;

pmt$queue_status = record
  connections: pmt$connected_tasks_per_queue,
  messages: pmt$messages_per_queue,
  waiting_tasks: pmt$connected_tasks_per_queue,
recend;

pmt$queued_segment = record {* not supported in R1}
  case kind: pmt$queued_segment_kind of
  = pmc$message_pointer =
    pointer: ^cell,
  = pmc$message_heap_pointer =
    heap_pointer: ^HEAP ( * ),
  = pmc$message_sequence_pointer =
    sequence_pointer: ^SEQ ( * ),
  casend,
recend;

pmt$queued_segment_kind = (pmc$message_pointer,
  pmc$message_heap_pointer,
  pmc$message_sequence_pointer);

pmt$queues_per_job = 0 .. pmc$max_queues_per_job;

pmt$segments_per_message = 1 ..
  pmc$max_segs_per_message;

pmt$sense_switches = set OF 1 .. 8;

pmt$standard_selection =
  (pmc$execute_standard_procedure,
  pmc$inhibit_standard_procedure);
```

Stack Frame Save Area D

A stack frame is the space allocated within a task stack to store the environment of a procedure and the contents of its local variables.

A stack frame has the format shown in figure D-1. Figure D-2 shows the format of the P register in the first word of the stack frame. Figure D-3 is the CYBIL declaration of the OST$STACK_FRAME_SAVE_AREA. Table D-1 describes the content of a stack frame save area.

A task is allocated stack space when it is initiated. The first frame of a task stack is that of the system task initiation procedure. When the initiation procedure calls the starting procedure of the program, a stack frame for that procedure is allocated on the stack. Subsequently, during the task, whenever a procedure is called, a stack frame is allocated for the procedure.

When a procedure completes and returns to its caller, its stack frame is removed from the stack. If the task completes normally via the starting procedure returning to its caller, the starting procedure frame is removed from the stack. If the task terminates by calling the PMP$EXIT or PMP$ABORT procedure, each frame of the stack is removed, in succession, without completion of the procedure associated with the stack frame. However, if a block exit processing condition handler is associated with a frame, the condition handler is executed before the frame is removed.

```
TYPE
  ost$stack_frame_save_area = record
    minimum_save_area: ost$minimum_save_area,
    undefined: 0 .. 0ffff(16),
    a3: ^cell,
    user_condition_register: ost$user_conditions,
    a4: ^cell,
    monitor_condition_register:
      ost$monitor_conditions,
    a5: ^cell,
    a_registers: array[6 .. 0f(16)] OF record
      undefined: 0 .. 0ffff(16),
      a_register: ^cell,
    recend,
    x_registers: array [ost$register_number] OF
      ost$x_register,
  recend;

TYPE
  ost$minimum_save_area = packed record
    p_register: ost$p_register,
    vmid: ost$virtual_machine_identifier,
    undefined: 0 .. 0fff(16),
    a0_dynamic_space_pointer: ^cell,
    frame_descriptor: ost$frame_descriptor,
    a1_current_stack_frame: ^cell,
    user_mask: ost$user_conditions,
    a2_previous_save_area:
      ^ost$stack_frame_save_area,
  recend;

TYPE
  ost$frame_descriptor = packed record
    critical_frame_flag; boolean,
    on_condition_flag: boolean,
    undefined: 0 .. 3(16),
    x_starting: ost$register_number,
    a_terminating: ost$register_number,
    x_terminating: ost$register_number,
  recend;

*copyc OSD$REGISTERS
*copyc OSD$CONDITIONS
*copyc OST$VIRTUAL_MACHINE_IDENTIFIER
```

Figure D-3. Stack Frame Save Area Type Declaration

Table D-1. Stack Frame Save Area (Type IST$STACK_FRAME_ SAVE AREA) *(Continued)*

Field	Content
frame_ descriptor	Packed record (type OST$FRAME_DESCRIPTOR). critical_frame_flag Boolean. on_condition_flag Boolean. undefined 0 through 3 hexadecimal. x_starting Type OST$REGISTER_NUMBER, integer. a_terminating Type OST$REGISTER_NUMBER, integer. x_terminating TYPE OST$REGISTER_NUMBER, integer.
a1_current_ stack_frame	Current stack frame pointer (^ cell).
user_mask	Set of user conditions (type OST$USER_CONDITIONS, see table D-2).
a2_previous_ save_area	Pointer to previous stack frame save area (type ^ OST$STACK_FRAME_SAVE_AREA).

(Continued)

Table D-2. User Conditions (OST$USER_CONDITIONS)

Following are the identifiers and meanings for OST$USER_CONDITIONS.

OSC$PRIVILEGED_INSTRUCTION
Improper attempt to execute a privileged instruction.

OSC$UNIMPLEMENTED_INSTRUCTION
Instruction code not implemented for this processor.

OSC$FREE_FLAG
Notification to process that an event occurred while it was not in active execution.

OSC$PROCESS_INTERVAL_TIMER
Process interval timer decremented to zero.

OSC$INTER_RING_POP
Attempted to pop stack frame from another ring.

OSC$CRITICAL_FRAME_FLAG
Attempted to return from a critical stack frame.

OSC$KEYPOINT
Keypoint instruction executed.

OSC$DIVIDE_FAULT
Error in divide operation.

OSC$DEBUG
Debug interrupt.

OSC$ARITHMETIC_OVERFLOW
Arithmetic overflow error.

OSC$EXPONENT_OVERFLOW
Exponent overflow error.

OSC$EXPONENT_UNDERFLOW
Exponent underflow error.

OSC$FP_SIGNIFICANCE_LOSS
Floating point significance loss.

OSC$FP_INDEFINITE
Floating point indefinite error.

OSC$ARITHMETIC_SIGNIFICANCE
Arithmetic significance loss.

OSC$INVALID_BDP_DATA
Invalid BDP data error.

Index

Index

CYBIL for NOS/VE, System Interface 60464115 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?	**How Do You Use This Manual?**	**Which Do You Also Have?**
☐ Manager	☐ As an Overview	☐ Any SCL Manuals
☐ Systems Analyst or Programmer	☐ To Learn the Product/System	☐ CYBIL File Interface
☐ Applications Programmer	☐ For Comprehensive Reference	☐ CYBIL Language Definition
☐ Operator	☐ For Quick Look-up	
☐ Other ________		

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	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple ☐ Too complex)
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ☐ Yes ☐ No

Continue on other side

From:

Name ________ Company ________

Address ________ Date ________

________ Phone No. ________

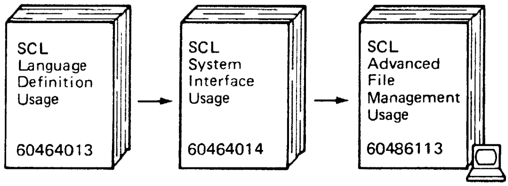
____

Please send program listing and output if applicable to your comment.

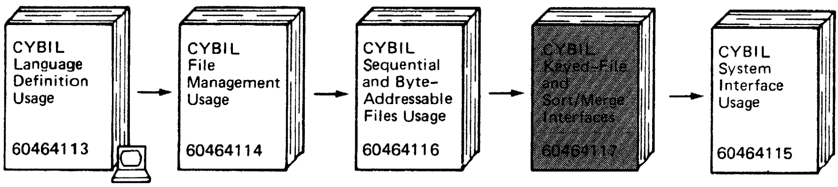
(Continued from inside front cover)

Related Manuals

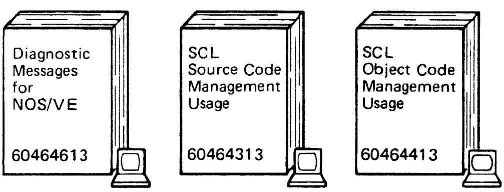
Background (Access as Needed):



CYBIL Manual Set:



Additional References:



→ Indicates the reading sequence.

Indicates an online version of the manual is available.

CYBIL Manual Set

This manual belongs to the CYBIL manual set. Besides this manual, the CYBIL manual set is composed of these manuals:

CYBIL Language Definition

Contains the complete language specification for CYBIL, the NOS/VE implementation language, and an explaination of the Debug utility as used with CYBIL.

CYBIL File Management

Describes the procedure calls that interface between a CYBIL program and the NOS/VE file system. It describes local file management and the assignment of files to device classes with a chapter describing each device class. It also describes file attribute definition and file opening and closing.

CYBIL Sequential and Byte Addressable Files

Describes the procedure calls that allow a CYBIL program to read and write sequential and byte addressable files. It describes both segment access and record access.

CYBIL System Interface

Describes system-defined CYBIL procedures that serve as the interface between a program and non-I/O system capabilities. It describes program management, condition processing, interstate communication, and system command language (SCL) calls.

▌	Vertical bars in the margin indicate changes or additions to the text from the previous revision.
●	A dot next to the page number indicates that a significant amount of text (or the entire page) has changed from the previous revision.

Ordering Manuals

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

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

Submitting Comments

The last page of this manual is a comment sheet. Please tell us about any errors you found in this manual and any problems you had using it.

If the comment sheet in this manual has been used, please send your comments to:

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

Please include this information with your comments:

The manual title, publication number, and revision level (for this manual: CYBIL Keyed-File and Sort/Merge Interfaces Usage, 60464117 B)

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

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

Also, if you have access to SOLVER, the CDC online facility for reporting problems, you can use it to submit comments about this manual. When it prompts you for a product identifier for your report, please specify AA8 when commenting on the keyed-file interface and SM8 when commenting on the Sort/Merge interface.

Procedure Deck Names

To use CYBIL program interface calls, you copy a deck for each procedure call you use. The deck has the same name as the procedure call.

For example, if your program uses the AMP$OPEN, AMP$GET_KEY, and AMP$CLOSE calls, it must use these three directives:

```
*COPYC AMP$OPEN
*COPYC AMP$GET_KEY
*COPYC AMP$CLOSE
```

Expanding Your Program

Before you compile a CYBIL program that uses program interface calls, you use SCU to expand the program, as follows:

1. You must begin with an existing source library file. If you do not have one, you can create an empty source library using the CREATE_SOURCE_LIBRARY command.

2. Start an SCU utility session, specifying a source library file.

3. Create one or more decks containing your program text.

4. Expand the decks containing your program text. Specify these two files as the alternate base libraries from which SCU copies the program interface decks:

   $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE
   $SYSTEM.COMMON.PSF$EXTERNAL_INTERFACE_SOURCE

5. End the SCU utility session.

This process gives you the expanded program text that can be compiled.

The following is a minimal command sequence that performs the preceding steps (numbered 1 through 5). It uses only temporary files and assumes your program text is on file $USER.PROGRAM_TEXT. (/, sc/, and sc../ are system prompts; you do not enter them.)

1. `/create_source_library result=temporary_library`

2. `/scu base=temporary_library`

3. ```
   sc/create_deck deck=temporary_deck ..
   sc../modification=temporary_modification source=$user.program_text
   ```

4. ```
   sc/expand_deck deck=temporary_deck ..
   sc../alternate_base=($system.cybil.osf$program_interface, ..
   sc../$system.common.psf$external_interface_source)
   ```

5. `sc/quit write_library=no`

Status Checking

The last parameter on every program interface call is the status parameter. You must specify a status variable (type OST$STATUS) as the last parameter on a call. When the procedure completes, it returns its completion status in the specified status variable.

You can specify an error-exit procedure to process errors returned by file interface procedures. (It does not process Sort/Merge errors.) The error-exit procedure is specified by the error_exit_name or error_exit_procedure file attribute.

If an error-exit procedure is specified for an instance of open, a file interface procedure calls the error-exit procedure when it returns abnormal status. The abnormal status is passed to the error-exit procedure which, in turn, passes its completion status to the status variable specified on the call.

An error-exit procedure is effective only while the file is open. It is not effective for AMP$OPEN or AMP$CLOSE calls. For these calls, and for files without error-exit procedures, you must check the contents of the status variable after the call to determine if the call completed successfully.

A status record is returned in the status variable. If the NORMAL field of the status record is TRUE, the procedure completed normally. If the NORMAL field is FALSE, the procedure completed abnormally.

For example, these lines show an AMP$OPEN call and the status check following the call:

```
AMP$OPEN ( lfn, AMC$RECORD_ACCESS, NIL, fid, status );
IF NOT status.NORMAL THEN
   PMP$EXIT( status );
IFEND;
```

For the PMP$EXIT call description and additional information on condition handling, see the CYBIL System Interface manual. A more complete example of status variable processing is given by the p#inspect_status_variable and p#display_status_variable procedures in appendix E.

Exception Condition Information

When the procedure completes abnormally, the procedure returns additional information about the exception condition (the error) that occurred. The following variant fields of the OST$STATUS record return condition information when the key field, NORMAL, is FALSE:

System Naming Convention

In general, all CYBIL program interface identifiers follow a system naming convention as follows:

idx$name

id — Two characters identifying the process that uses the identifier. (These are the same process identifiers returned in the IDENTIFIER field of the status record.)

x — Character indicating the type of CYBIL element identified. These are the element types:

c	Constant
d	Declaration of multiple or complex types
e	Error condition
f	File
i	Inline text or code
k	Keypoint or keyword
m	Module
p	Procedure
s	Section
t	Type
v	Variable
x	Element with XDCL attribute

$ — The $ character indicates that CDC defined the identifier.

NOTE

To avoid redefining a CDC identifier, do not use the $ character in identifiers that you define.

name — A string describing the purpose of the element referenced by the identifier.

For example, the identifier AMP$CREATE_KEY_DEFINITION follows the naming convention:

- Its process identifier is AM (Access Method).
- It identifies a procedure (P).
- It is a CDC-defined identifier ($).
- Its purpose is the creation of an alternate-key definition (CREATE_KEY_DEFINITION)

Indexed-Sequential File Organization

The indexed-sequential file organization allows content addressing of records; that is, you can directly access a record by the contents of one or more fields of data in the record. The fields of data by which a record is addressed are its key fields, and the contents of those fields are its key values.

An indexed-sequential file always has a primary key. (It can also have one or more alternate keys as described in the Alternate Keys section of this chapter.)

Each primary-key value is unique within the file; there can be no duplicate primary-key values in a file.

The indexed-sequential file organization is used only when you can assign a unique value to each record stored in the file. This unique value is usually a field of data within the record (an embedded key), although it can be a value assigned to the record and not included in the record data (a nonembedded key).

For example, the primary key for an employee file could be the employee's name. However, because two employees could have the same name, it is better to assign a unique identification number to each employee and use that number as the primary key for the file.

The indexed-sequential file organization should be used if a requirement exists to read file records both sequentially and randomly. For example, the records in an employee file could be read sequentially to produce a listing of all employees or read randomly to update individual records.

When an indexed-sequential file is read sequentially, its records are accessed in ascending order by key value. The order is kept even when new records are added to the file. For example, if an employee file is read sequentially using its primary key (the employee identification number), the records are read in ascending order by their identification number.

Indexed-Sequential File Structure

This section gives a general description of the indexed-sequential structure. You can use indexed-sequential files without knowing their structure. However, if you understand the indexed-sequential structure and how it grows, you can create more efficient indexed-sequential files by specifying appropriate values for structural parameters.

The internal structure of an indexed-sequential file is designed to provide both random and sequential access to the data records in the file. File space is divided into blocks, all the same size.

Let's suppose you request to read randomly the record with key value 6. When the record is read, these steps are performed:

1. The index records are searched to find the index record whose range of key values includes the key value 6.

2. After the correct index record (the second one) is found, the search for the record continues with the data block to which the second index record points.

3. The second data block is searched for the record with key value 6. When the record is found, its data is returned to the requestor.

Next, suppose you request that all records in the file shown in figure I-1-1 be read sequentially. These steps are performed:

1. The first index record is read to find the first data block.

2. The records from the first data block are read in order.

3. The second index record is read to find the second data block.

4. The records from the second data block are read in order.

5. The sequential read ends because there are no more index records and, so, no more data blocks to read.

This process reads the records in key-value order because both the index records and the data records are kept in key-value order.

Data-Block Split

Usually, a block has some empty space, called padding, that was left empty so that additional records could be written later to the block. Suppose, as shown in figure I-1-2, that a data block has been filled, a new record is to be written, and its key value is within the range of key values of the records in the full data block. For the file structure to be maintained, the data block must be split.

When a data-block split occurs, records in the data block whose key values are less than the key value of 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 relative amount of empty space in the blocks and the size of the new record. If the new record does not fit in either block, another new block is created and the new record is put into that block.

Index Levels

As with data blocks, index blocks are also initially created with some empty space (index-block padding). However, for each new data block created due to a data-block split, another index record must be created. With the addition of many data records, the initial index block becomes full. When the index block is full, the next data-block split causes an index-block split.

As shown in figure I-1-3, when the initial index block splits, it causes the creation of another index level.

The index levels are numbered from the top down as index level 0, index level 1, and so forth. Index level 0 always has only one index block; it is always the starting point for an index search.

The index block at an upper level contains an index record for each index block at the next lower level. For example, the index block at level 0 contains an index record for each index block at level 1.

A search for a data record requires an index-block search at each index level. For example, the level-0 search finds the index record that points to the appropriate level-1 index block. If the file has only two index levels, the level 1 search finds the index record that points to the appropriate data block.

As you can see, the addition of another index level increases the time required to find an individual data record.

Index levels can be added up to the index-level limit of 15 levels. This sets a limit on the number of records in the file.

The index-level limit is reached when addition of another record to the file would require creation of another index level, but 15 index levels already exist in the file. When this happens, the index-level-overflow flag is set and no more records can be added to the file.

(Continued)

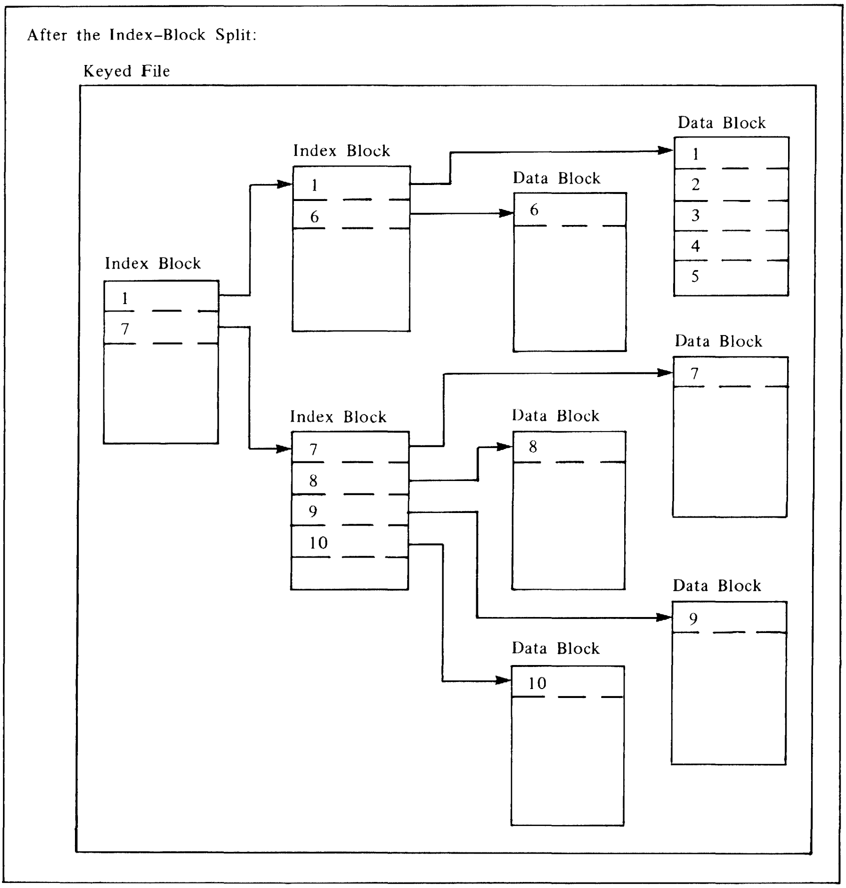


Figure I-1-3. Index-Block Split

Direct-Access File Organization

The direct-access file organization is like the indexed-sequential file organization in its use of a primary key. You define the primary key for the file when you create the file. It can be a field embedded in the record or a nonembedded value. Each primary-key value in the file must be unique; the file can contain no duplicate primary-key values.

Like an indexed-sequential file, a direct-access file can have alternate keys. An alternate key for a direct-access file is the same as an alternate key for an indexed-sequential file. Alternate keys are described later in this chapter.

Like indexed-sequential file records, you must specify the primary-key value when writing or deleting a direct-access file record. Similarly, you must specify either a primary-key value or an alternate-key value to read a direct-access file record.

Direct-access and indexed-sequential files differ in the ordering of records in the file:

- When records are read sequentially from an indexed-sequential file, the records are returned in order, sorted by primary-key value.
- When records are read sequentially from a direct-access file, the records are returned unordered.

In general, random record access is faster for the direct-access file organization than for the indexed-sequential file organization. This is because the direct-access file organization determines the location of a record directly from its primary-key value. (In indexed-sequential files, a record can be found only after a search at each index level.)

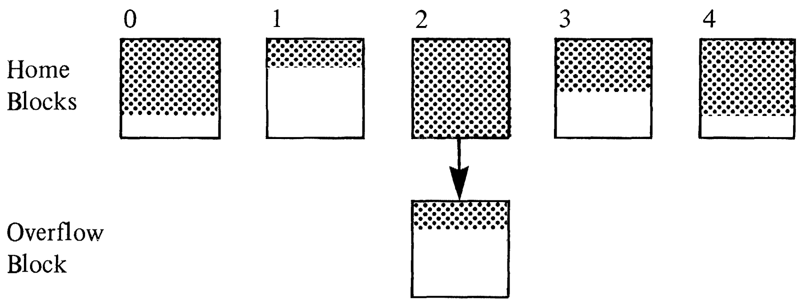
Direct-Access File Structure

The direct-access file structure is designed to locate each record directly by its primary-key value. The primary-key value directly specifies the file block containing the record.

File space in a direct access file is divided into equal-size blocks. Initially, all blocks in the file are home blocks (as opposed to overflow blocks).

When a record is written to a direct-access file, its primary-key value is hashed to produce the number of the home block in which the record is written. If the home block does not contain enough empty space for the new record, the record is written to an overflow block.

At this point, a record is to be written with primary-key value ABC. Hashing of the value ABC produces block number 2, but there is insufficient space for the record in home block 2 so it is written in an overflow block.



Later, to read the record with primary-key value ABC, the primary-key value is hashed to produce block number 2. Home block 2 is searched for primary-key value ABC. When it is not found in the home block, the search continues in the overflow block until the record is found.

An ideal direct-access file structure has these characteristics:

- Sufficient home blocks are allocated and records are uniformly distributed among the home blocks so as to avoid overflow.
- Each block contains a limited number of records so as to minimize the search time in each block.
- The number of home blocks is not so large that the file contains excessive unused space.

These characteristics are determined by the file attribute values specified when the file is created. You must specify the initial_home_block_count and can optionally specify the max_block_length and the hashing_procedure_name attributes. (The attributes are described in chapter I-2.)

One other characteristic to be considered when selecting the number of home blocks is the loading factor. The loading factor is the percentage of block space used. To allow for less-than-uniform distribution of records in the home blocks, the loading factor should be no greater than 90%.

To illustrate, suppose the direct access file is to contain 10,000 80-byte records (80,000 bytes of record data). Using a block size of 4096 bytes, 20 home blocks would be sufficient if the hashing procedure could guarantee uniform distribution of the records in the home blocks. This would result in a loading factor of nearly 98% (80,000 divided by 81,920). However, because uniform distribution should not be expected, the number of home blocks allocated should be at least 22 (for a loading factor of 89%). (It is also recommended that the home block count be a prime number; thus, 23 would be a better home block count for the file in this example.)

The system divides the value it receives from the hashing procedure by the number of home blocks and uses the remainder as the home block number. For example, if the number of blocks is 97, it divides the hashed value by 97 and uses the remainder (an integer from 0 through 96) as the home block number. A more uniform distribution of records is expected if the number of home blocks is a prime number.

Direct-Access Primary Keys

In general, the primary key of a direct-access file has the same characteristics as the primary key of an indexed-sequential file. You specify whether the primary key is embedded or nonembedded, its position (if the key is embedded), and the key length. However, a key_type attribute value specified for a direct-access file is ignored; the key_type attribute for a direct-access file is always uncollated.

Unlike an indexed-sequential file, sequential access calls to a direct-access file while the primary-key is selected do not return the file records sorted by primary-key value. The calls return records according to their physical location in the direct-access file. Records within each block are ordered according to the default ASCII collating sequence, but the blocks are not ordered by primary-key values.

Direct-access file records can be accessed in order if one or more alternate keys are defined for the file. The alternate index keeps the alternate-key values in sorted order. Sequential access calls while an alternate key is selected return records in the order provided by the alternate index.

If appropriate, you could define an alternate key for the same field as an embedded primary key. In this way, you could access direct-access file records in primary-key value order.

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. For example, the license numbers of several cars owned by one person as follows:

Data Record: `R. Petty  1 LB AU  2ASM451  ELK 592`

Alternate Index:

Alternate Key Value	Primary Key Values
1 LB AU	R. Petty
2ASM451	R. Petty
ELK 592	R. Petty

The Alternate Index

The index for the primary key was described earlier in this chapter. Each alternate key defined for a file has its own index.

An alternate index contains index records, each of which associates an 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 the key list for that alternate-key value.

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 for the alternate-key value to access the data 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

The attributes of an alternate key are specified by its alternate-key definition.

These attributes are required to define an alternate-key:

Key name
Key position
Key length

An alternate key has a name so that it can be selected later for use. The alternate-key position and length define the alternate-key field within the record.

For example, suppose you write three records to the file in this order:

McDarrels	Hamburgers
Burger Duke	Hamburgers
Willys	Hamburgers

The following shows the resulting key list in primary-key order and in first-in-first-out order:

	Key Lists	
Alternate Key Value	**Ordered by Primary Key**	**First In First Out**
Hamburgers	Burger Duke	McDarrels
	McDarrels	Burger Duke
	Willys	Willys

Duplicate-Key Value Error Processing

If duplicate values are not allowed and a duplicate is found in a record about to be written to the file, the record is not written to the file and a trivial error (status AAE$DUPLICATE_ALTERNATE_KEY) is returned.

A trivial error (status AAE$UNEXPECTED_DUP_ENCOUNTERED) also occurs if a duplicate value is found while a new alternate index is being created. However, the record containing the duplicate value cannot be discarded, because it is already in the file. Subsequent processing depends on whether incrementing the trivial-error count causes the count to exceed the trivial-error limit as set by the user.

- If the trivial-error limit is not exceeded, the apply operation redefines the alternate key being applied to allow duplicates, ordered by primary-key value, discards the partially built index, and builds the redefined index.
- If the trivial-error limit is reached, the apply operation returns the status condition AAE$DUPLICATE_KEY_LIMIT and removes all alternate indexes it has created. (Deleted indexes are not restored.)

In either case, a message describing the action taken is written to the $ERRORS file.

Sparse-Key Control

You can use sparse-key control to create an alternate index that includes or excludes records depending on the character in a specific position in the record.

For example, suppose a student file has a one-character code indicating the student's class. To get a mailing list for juniors and seniors only, you could define an alternate index controlled by the class code.

To specify sparse-key control, you specify three values:

Value	Example
Sparse-key control position	Position of the class code in the record
Sparse-key control characters	Junior and senior class code characters
Sparse-key control effect (Indicates whether the alternate-key value should be included or excluded if the sparse-key character matches)	Included if the class code indicates a junior or senior record

Assume that the sparse-key control position is the first character after the name field and that the junior and senior class codes are 3 and 4. If the following records are copied to the file, the first three records are included in the alternate index, but not the last record.

```
Louis Skolnik     4
Gilbert Sullivan  4
Elliot Wermzer    3
Judy Manhasset    2
```

The sparse-key control position must be within the minimum record length. If you specify sparse-key control for an alternate key, the alternate-key field or fields need not be within the minimum record length.

A nonfatal (trivial) error (status AAE$SPARSE_KEY_BEYOND_EOR) is returned if both of these conditions are true for a record:

- The character at the sparse_key_control_position indicates that the record should be included in the alternate index
- The record has no alternate-key value because the record ends before the alternate-key field

When an apply or write operation detects this error, it does not include the record in the alternate index. (A write operation does write the record to the file.)

Repeating Groups

The repeating-groups attribute allows a data record to contain more than one value for the same alternate key. This allows a primary-key value to be associated with more than one alternate-key value.

To specify an alternate-key field within a repeating group:

1. Specify the first alternate-key field by its key position, key length, and key type. All subsequent alternate-key fields have the same length and type as the first.

2. Specify repeating groups for the alternate key by specifying the repeating group length: that is, the distance from the beginning of the first instance of the alternate key to the beginning of the second instance of the alternate key in the record.

3. Specify the repeating-group count: that is, how many times the alternate key field repeats in the record.

You can specify that the repeating group repeats a fixed number of times or that it repeats until the end of the record.

- If the alternate-key field repeats a fixed number of times, all alternate-key fields must be within the minimum record length.

- If the alternate-key field repeats to the end of the record, the minimum record length imposes no restriction. The system stores as many alternate-key values as the record length allows.

Repeating groups cannot be used with concatenated keys or when duplicate-key values are allowed and ordered first-in-first-out.

For example, suppose each record in a membership file lists the sports the member enjoys and his or her years of experience as follows (columns are counted from zero):

Field: Sports and Sports Experience

Columns: Variable number of 2-field pairs beginning at column 75 The Sports field is 10 characters followed by a 2-digit Sports Experience field

Type: ASCII characters

Nested Files

A nested file is a file structure defined within a NOS/VE file cycle. It is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

The keyed-file interface provides nested files so as to extend the NOS/VE limit on the number of files a task can use. All nested files defined in a file share the same memory segment. This provides effective memory use when the nested files are much smaller than the segment size limit (2^{32} bytes).

The keyed-file interface creates the initial nested file (named $MAIN_FILE) when it creates the keyed file. It uses $MAIN_FILE as the default nested file; other nested files are used only when explicitly selected.

An AMP$CREATE_NESTED_FILE call can create a nested file (in addition to the default nested file $MAIN_FILE). The call defines the attributes applicable to the nested file only. These include its:

File organization

Record attributes, including its record type and its minimum and maximum record lengths

Primary-key attributes, including its key position, key length, key type, and collation table

Structural attributes applicable to the file organization

All other file attributes apply to all nested files in a keyed file. The RECORD_LIMIT attribute specifies the maximum number of records in each nested file. For more information on attributes, see Creating a Keyed File later in chapter I-2.

Each alternate-key definition applies to only one nested file. To define an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then define the alternate key. Similarly, to select an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then select the alternate key.

File_Organization Attribute

To create a keyed file, you specify a keyed-file organization as the file_ organization attribute. Currently, the keyed-file organizations are indexed-sequential and direct-access.

To specify indexed-sequential file organization, you initialize an attribute record as follows:

```
[AMC$FILE_ORGANIZATION, AMC$INDEXED_SEQUENTIAL]
```

To specify direct-access file organization, you initialize an attribute record as follows:

```
[AMC$FILE_ORGANIZATION, AMC$DIRECT_ACCESS]
```

The other keyed-file attributes define record attributes, primary key attributes, file structure attributes, and processing attributes.

Record Attributes

These attributes describe the data records to be written to the keyed file.

NOTE

The record attributes are all preserved attributes, that is, the attribute value is stored with the file when the file is first opened and cannot be changed thereafter.

The following lists the CYBIL attribute identifier (AMC$xxx) followed by the valid attribute values:

AMC$RECORD_TYPE

Record type: AMC$FIXED, AMC$VARIABLE, or AMC$UNDEFINED. The default is AMC$UNDEFINED.

AMC$MAX_RECORD_LENGTH

Maximum number of bytes in a data record (from 1 through 65497). You must specify a value for this attribute when defining a keyed file.

AMC$KEY_POSITION

Position of the leftmost byte in the primary key (specified only if the key is embedded). The byte positions in a record are numbered from the left, beginning with 0. The default is 0.

AMC$KEY_TYPE

Primary key type: AMC$UNCOLLATED_KEY, AMC$INTEGER_KEY, or AMC$COLLATED_KEY. The default is AMC$UNCOLLATED_KEY.

For direct-access files, any value specified for the key_type attribute is ignored. The key_type for a direct-access file is always uncollated.

AMC$COLLATE_TABLE_NAME

Name of the collating sequence by which collated keys are ordered (required if the key_type is collated).

The name can be the name of a NOS/VE predefined collating sequence or, for a user-defined collating sequence, the name of an entry point in an object library. See appendix D for more information.

File Structure Attributes

These attributes affect the internal file structure. Keyed-file structure is described in chapter I-2.

The first group of attributes applies to all keyed-file organizations; the groups that follow each apply to one keyed-file organization only.

NOTE

The file structure attributes are all preserved attributes. That is, the attribute value is stored with the file when the file is first opened and (except for record_limit) cannot be changed thereafter.

Block Length Guideline Attributes

NOTE

The following attributes do not set limits; their values are used only as guidelines for determining the block length when the file is created.

AMC$AVERAGE_RECORD_LENGTH

Estimated median record length, in bytes, of the data records to be stored in the file. (The length should not include a nonembedded key.)

If you omit this parameter, the system uses the arithmetic mean between the maximum and minimum record lengths in its calculation of the block size.

AMC$ESTIMATED_RECORD_COUNT

Estimated number of data records to be stored in the file. If you do not define this attribute, the system uses in its calculation of the block size either the AMC$RECORD_LIMIT value, or if that attribute is not defined, the value 100,000.

AMC$INDEX_LEVELS

Target number of index levels for the file (0 through 15). The default value is 2.

This attribute applies only to indexed-sequential files.

AMC$RECORDS_PER_BLOCK

Estimated number of data records to be stored in each data block. If you do not define this attribute, the system uses the value 2 in its calculation of the block size.

AMC$HASHING_PROCEDURE_NAME

Pointer to a record identifying the hashing procedure to be executed with this file (^amt$hashing_procedure_name). The record has these fields:

NAME	Entry point name of the hashing procedure (pmt$program_name). All letters in the name must be specified as uppercase.
OBJECT_LIBRARY	File path to the object library containing the hashing procedure (amt$path_name, 256-character string). This feature is currently unimplemented; specify OSC$NULL_NAME as the field value.

The default hashing procedure is the one provided by the system, entry point AMP$SYSTEM_HASHING_PROCEDURE.

If a hashing procedure other than the default is specified, it must be a procedure declared with the XDCL attribute within the global library set of the job or defined within the task. The hashing procedure must be available whenever the file is used; otherwise, AMP$OPEN returns the condition aae$cant_load_hash_routine.

Processing Attributes

These attributes set keyed-file processing options.

NOTE

The forced_write and lock_expiration_time attributes are preserved attributes, but their values can be changed by the CHANGE_FILE_ATTRIBUTES command. For more information, see the SCL System Interface Usage manual.

The error_limit and message_control attributes are temporary attributes; their values can be changed each time the file is opened.

Writing Records

Records can be written to a keyed file opened with at least append access. (If alternate keys are defined for the file, it must be opened with modify, append, and shorten access.)

You can write records to a new keyed file using either AMP$PUT_KEY or AMP$PUT_NEXT calls. Use of AMP$PUT_KEY calls is recommended for writing keyed files. AMP$PUT_NEXT should be used only if a common interface for writing records, regardless of file organization, is required.

NOTE

An AMP$PUT_NEXT call cannot specify a key value. When the keyed file has a nonembedded primary key, AMP$PUT_NEXT takes the key value from the beginning of the working storage area. It stores the first key_length bytes as the nonembedded primary-key value and the rest of the data as the record.

In general, pre-sorting records to be written to an indexed-sequential file can result in a smaller file and less time required for writing the records. Your program can use NOS/VE Sort/Merge to sort records as described in part II of this manual.

For an indexed-sequential file with an embedded primary key, you could use NOS/VE Sort/Merge calls to write the original set of records to the file. (NOS/VE Sort/Merge calls are described in part II of this manual.) The Sort/Merge specification must define the primary-key field as the major sort key.

Re-creating a Keyed File

As described earlier, the initial keyed-file structure is created when the file is first opened using the file structure attribute values defined for the file. As records are added, replaced, and deleted in the file, the file structure may become inefficient. When this becomes evident, you should re-create the file to improve the efficiency of its structure.

The evidence of an inefficient file structure differs depending on the keyed-file organization.

Using a Keyed File

To process an existing keyed file, a CYBIL program performs these steps:

1. Specifies temporary attribute values to be used by this instance of open and preserved attribute values to be verified against the attribute values stored with the file (AMP$FILE and AMP$OPEN).
2. Opens the keyed file for record access (AMP$OPEN).
3. Performs the intended file operations.
4. Closes the file (AMP$CLOSE).

The following file operations can be performed on an existing keyed file (assuming the file has been opened with the required access modes):

- Position the file (AMPGET_KEY, AMPREWIND, AMP$SKIP, and AMP$START).
- Read records randomly by key value (AMP$GET_KEY).
- Read records sequentially by position (AMP$GET_NEXT_KEY and AMP$GET_NEXT).
- Write records (AMPPUT_KEY, AMPPUT_NEXT, and AMP$PUTREP).
- Delete records (AMP$DELETE_KEY).
- Replace existing records (AMP$REPLACE_KEY and AMP$PUTREP).
- Lock key values (AMP$LOCK_KEY, AMP$GET_LOCK_KEYED_RECORD, AMP$GET_LOCK_NEXT_KEYED_RECORD, and AMP$LOCK_FILE).
- Unlock key values (AMP$UNLOCK_KEY and AMP$UNLOCK_FILE).
- Define, delete, and select nested files (AMP$CREATE_NESTED_FILE, AMP$DELETE_NESTED_FILE, AMP$GET_NESTED_FILE_DEFINITIONS, and AMP$SELECT_NESTED_FILE).
- Define, delete, and select alternate keys as described later in this chapter.

Depending on the value of the forced_write attribute, the system might not write modified blocks to mass storage immediately after the modification. You can call AMP$FLUSH any time after the file is opened to write the part of the file in memory to mass storage. Execution of the AMP$FLUSH call does not change the position of the file.

Positioning an Indexed-Sequential File by Major Key

The AMP$START, AMP$GET_KEY, and AMP$GET_LOCK_KEYED_RECORD calls have a major_key_length parameter. This parameter allows a call to position an indexed-sequential 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 value at the specified key_location is ABCDEF and the major_key_length parameter value is 2. The major-key value, therefore, 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 value 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.

The major_key_length parameter is ignored on direct-access file calls.

Positioning an Indexed-Sequential File by Key Relation

The AMPGET_KEY, AMPGET_LOCK_KEYED_RECORD, and AMP$START calls have a key_relation parameter. This parameter allows a call to position an indexed-sequential file even if the specified key value does not exist in the file.

The key_relation parameter specifies the relation to be satisfied between the specified key value and the key value of the record at which the file is positioned. The relation can be equal, greater than or equal, or greater than.

For example, suppose the specified key value is ABC.

- If the specified key_relation is equal, the call must find a record whose key value matches ABC. If such a record is not found, the call returns an abnormal completion status.

- If the specified key_relation is greater than or equal to, the first key value found that is greater than or equal to ABC satisfies the relation. If the relation cannot be satisfied, the file is left positioned at its end-of-information.

Sequential Access for Direct-Access Files

Records are not stored in sorted order by primary-key value in direct-access files as they are in indexed-sequential files. Thus, sequential access is appropriate only:

- When an alternate key is selected
- When a primary key is selected and all records in the file are to be read

A sequential pass through a direct-access file is valid only when no update operation intervenes. An intervening update operation could cause the sequential pass to miss records. (Sequential access to a direct-access file is done by physical position in the file; an update operation could change the record locations.)

To provide effective sequential access, the keyed-file interface imposes these restrictions on sequential access to direct-access files:

- When the primary key is selected, AMP$GET_LOCK_NEXT_KEY, AMP$GET_NEXT_KEY and AMP$GET_NEXT calls are valid only when the direct-access file has been attached for exclusive access (no share modes allowed).

  When the primary key is selected and the file attachment allows sharing, a sequential get call returns the condition aae$cant_da_getn_if_shared.

- When the primary key is selected, a program cannot intermix sequential access calls and update operations. (The only update operation allowed is the replacement of a record with another record of the same length.)

  When the primary key is selected and an update operation has been performed, the program must rewind the file before beginning a sequential pass of the direct-access file. Otherwise, a sequential get call returns the condition aae$cant_da_getn_after_put.

You can intermix sequential access (get_next) calls and AMP$GET_KEY calls. An AMP$GET_KEY call does not change the file position used by get_next calls.

Keyed-File Sharing

A NOS/VE keyed file can be accessed with or without potential sharing of the file. A keyed file is shared when multiple concurrent instances of open of the file exist.

The potential for sharing determines whether NOS/VE must safeguard the keyed-file structure for multiple users:

- While a keyed file could be shared, NOS/VE performs internal locking operations to maintain the integrity of the file structure.
- While a keyed file cannot be shared, the overhead required to maintain file integrity is not needed, resulting in better file access performance.

File access is controlled by the set of access modes in effect for the file. File sharing is controlled by the set of share modes in effect. The use of access modes and share modes for NOS/VE files in general is described in the SCL System Interface and CYBIL File Management manuals; access mode and share mode use for keyed files is described here.

To see the access modes and share modes currently in effect for a file, enter this SCL command (specifying the file name or file reference):

```
Display_File_Attributes, File=file, ..
  Display_Options=(Access_Modes, Global_Share_Modes)
```

The Access_Modes set is the set of access modes currently in effect. It is contained in the Global_Access_Modes set (the set of all available access modes as determined when the file is created or attached). When the file is created or attached, the Access_Modes and Global_Access_Modes values sets are the same. However, the Access_Modes set can be restricted to a subset of the Global_Access_Modes by a SET_FILE_ATTRIBUTES command or AMP$FILE or AMP$OPEN call. Keyed-file sharing is affected only by the Access_Modes set; the Global_Access_Modes set only indicates the possible values of the Access_Modes set.

The Global_Share_Modes set is the set of share modes currently in effect. It is determined when the file is created or attached; you cannot change the Global_Share_Modes using SET_FILE_ATTRIBUTES commands or AMP$FILE or AMP$OPEN calls.

AMP$GET_FILE_ATTRIBUTES and AMP$FETCH calls in a CYBIL program can fetch the Access_Modes and Global_Share_Modes sets.

In the first situation, no locking is needed because no sharing is allowed. In the second situation, no locking is needed because the data cannot change. When no locking is needed, no setting of locks or checking for locks is done and performance improves.

NOTE

For best performance when using a keyed file, check that the share modes allowed are no more than those required. If possible, allow no sharing of the file.

In general, when the file can be shared (the Global_Share_Modes value is not none) and either the Access_Modes or the Global_Share_Modes include shorten or append access, locking is needed. The following examples show two situations in which locking is not needed and a third situation in which it is needed.

1. When reading a keyed file, it is recommended that you request modify access so that read statistics can be recorded in the file. Because modify is one of the write access modes, no other instances of open can access the file while you read it (if you do not explicitly specify Share_Modes). For example:

   ```
   /attach_file, $user.keyed_file, access_modes=(read, modify)
   /display_file_attributes, keyed_file, ..
   ../display_options=(access_modes, global_share_modes)
   Access_Mode                    : (read, modify)
   Global_Share_Mode              : none
   ```

   In this case, because no sharing is allowed, no locking is performed and performance is at its best.

2. Next, to allow other users to read the keyed file and maintain accurate read statistics, you explicitly specify the Share_Modes as read and modify:

   ```
   /attach_file, $user.keyed_file, access_modes=(read, modify) ..
   ../share_modes=(read, modify)
   /display_file_attributes, keyed_file, ..
   ../display_options=(access_modes, global_share_modes)
   Access_Mode                    : (read, modify)
   Global_Share_Mode              : (read, modify)
   ```

   In this case, sharing is allowed, but the file data cannot be changed. So again, no locking is performed and performance is at its best.

The lock manager also processes requests to clear locks and keeps track of locks that have expired (as described later under Lock Expiration and Clearing).

NOTE

In general, when the discussion of locks in this manual describes two or more tasks requesting locks, the two or more tasks could actually be the same task with two or more instances of open of the same file. This is because a lock belongs to a particular instance of open and one task could be requesting locks for more than one instance of open.

Lock use is recommended for effective sharing of a keyed file. In fact, when more than one instance of open exists for a keyed file, NOS/VE requires that a task lock the record before it can replace or delete the record.

Lock use ensures that:

- Requests are processed in the sequence in which requests are issued.
- The operation is performed on the most up-to-date version.

Reasons for Locks

To illustrate the need for locks, the following sequence of events describes two tasks using the same file without locks.

1. Two tasks both read the same record containing the value 1.

File	Task A	Task B
1	1	1

2. One task adds 2 to the value and replaces the record, containing the value 3, in the file.

File	Task A	Task B
3	3	1

3. The other task adds 1 to the value and replaces the record, containing the value 2, in the file.

File	Task A	Task B
2	3	2

The work of one of the tasks has been overwritten.

Lock Intents

Each lock has a lock intent. The lock intent indicates why the task is requesting the lock.

When more than one instance of open exists for a keyed file, only the owner of an Exclusive_Access or Preserve_Access_and_Content lock on the record (or the file) can replace or delete the record. However, the replace or delete operation does not take place until no unexpired Preserve_Content locks exist for the record.

The following paragraphs describe the lock intents for record locks. (Lock intents for file locks are described later under File Locks.)

Exclusive_Access

- Used when the task intends to issue write or delete requests for the locked record.
- Denies all requests by other tasks to read, write, update, or delete the record or lock its key value.
- Allow requests by other tasks that position the file or perform operations only on alternate indexes.

Preserve_Access_and_Content

- Used when the task might issue write or delete requests for the locked record. Only one Preserve_Access_and_Content lock is allowed at a time for a record.
- Allows positioning and read requests by other tasks, but denies their write, replace, and delete requests.
- Allows Preserve_Content lock requests by other tasks, but denies their requests for Exclusive_Access and Preserve_Access_and_Content locks on the record.
- The owner of the Preserve_Access_and_Content lock can request a write, replace, or delete operation, but:
  - The write, replace, or delete operation does not begin until the conditions for an Exclusive_Access lock are met:
    - All read operations in progress for the record have completed.
    - All Preserve_Content locks for the record have expired or been cleared.
  - No read operations for the record can begin until the write, replace, or delete operation completes.

Waiting for a Lock

On a call that requests a lock, you specify whether the call should wait if the lock is unavailable. If you specify that the call should wait, it waits until the lock is available or a lock timeout period has passed. When the time period has passed, the call terminates with the condition aae$key_timeout.

The default timeout period is 60 seconds. However, each task can specify how long it waits for a lock by defining and initializing an SCL integer variable.

The timeout variable is named AAV$RESOLVE_TIME_LIMIT. You assign the variable the new waiting period in seconds (from 1 through 604,800,000 [1 week]).

For example, the following call executes the SCL command CREATE_VARIABLE to create the AAV$RESOLVE_TIME_LIMIT variable and assign it the value 45.

```
clp$scan_command_line('create_variable, AAV$RESOLVE_TIME_LIMIT, CAT
  kind=integer, value=45, scope=local', status);
```

(The CLP$SCAN_COMMAND_LINE call is described in the CYBIL System Interface manual.)

Lock Expiration and Clearing

An expired lock and a cleared lock are not the same:

- A cleared lock no longer exists; the lock manager has discarded it.
- An expired lock exists, but is no longer effective in preventing access by other tasks. However, an expired lock prevents file access by its owner (except to fetch or store attributes or access information). This is done so that the owner of the lock is notified of its expiration.

A lock is cleared when one of these events occurs:

- The task with the lock issues an unlock request for the lock.
- The task closes the instance of open to which the lock belongs.
- The request for the record lock specified automatic unlock, and the task issues any request for the instance of open (other than a call to fetch or store attributes or fetch access information).

In general, the automatic unlock occurs when the request is issued. The exception is for an update request for the locked record for which the lock is kept until the update operation completes.

7. Task 1 attempts to read record 2 in file 1, but instead the request terminates with a nonfatal error, notifying Task 1 that it has an expired lock. Task 1 must clear the expired lock before it can successfully request any record in file 1.

Notice that in the preceding example the lock would not have expired if the lock request had specified automatic unlock.

Expired Lock Conditions

The following nonfatal conditions can be returned for an expired lock:

`aae$key_expired_lock_exists`
: The operation failed due to a leftover expired lock.

`aae$auto_unlock_frustrated`
: A key value could not be automatically unlocked due to an expired lock.

`aae$key_expired_lock_exists`
: The key value could not be locked due to an expired lock.

`aae$expired_lock_interfered_1`
: A lock with a time limit could not be changed to a lock with no time limit due to an expired lock.

`aae$expired_lock_interfered_2`
: The first primary-key value in the key list for an alternate-key value could not be locked due to an expired lock. This status can be returned only if the alternate key allows duplicate values, ordered by primary key, and, while the task is waiting for the lock, another task inserts a primary-key value at the beginning of the key list.

File Locks

Your program should request a file lock when it needs locks on many keys at the same time.

A file lock is required when your program needs more than 1024 locks at a time because 1024 is the maximum number of locks allowed for an instance of open. An attempt to exceed this limit returns the nonfatal condition aae$too_many_keylocks.

The number of locks allowed also depends on the file_limit attribute value. The lock manager tracks all locks for a file in another file called the lock file (named AAF$DEPENDENCY_FILE). The lock file size cannot exceed 90% of the file_limit value and, if an operation would cause the lock file to be more than 50% full, the operation is not allowed to begin and the fatal condition aae$lock_file_crowded is returned.

In general, the rules for using file locks are the same as those for individual locks on primary-key values. The difference is that a file lock is a lock on all primary-key values in the nested file currently selected.

A nested file cannot be deleted while any locks exist for the nested file. Locks are not discarded even when another nested file is selected.

File Lock Intents

The effect of the lock intent of a file lock is as follows:

- Exclusive_Access

  Only the owner of the lock can access records in the nested file; all requests by nonowners are denied including all lock requests.

- Preserve_Access_and_Content

  Allows Preserve_Content locks (both key locks and file locks), but denies all Exclusive_Access and Preserve_Access_and_Content locks.

- Preserve_Content

  Allows any number of Preserve_Content locks and one Preserve_Access_and_Content lock for each primary-key value and for the nested file as a whole, but denies all Exclusive_Access lock requests.

Creating and Deleting Alternate Keys

To create or delete alternate keys, a CYBIL program performs these steps:

1. Opens the file, if it is not already open.

2. Issues 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, it issues an AMP$APPLY_KEY_DEFINITIONS call. Or, to discard the specified definitions and deletions, it issues an AMP$ABANDON_KEY_DEFINITIONS call.

A program can create alternate keys in a new file or in an existing file. The point at which you should create alternate keys depends upon how the alternate key handles duplicate values.

If the file data is expected to contain duplicate values for the alternate key and the duplicate values are to be ordered first-in-first-out, the alternate key must be defined before records are written to the file. Otherwise, when the alternate index is built, the duplicate values already existing in the file are ordered by primary-key value. Duplicate values added later are ordered first-in-first-out.

If duplicate key values are not allowed for the alternate key or the duplicate values are to be ordered by primary-key value, the alternate key should be defined after records are written to the file. Building the alternate index is more efficient when the records are already in sorted order. If the alternate index is updated as each record is written, the alternate index is built in random order. This takes much longer. The efficiency difference is even greater when the file has more than one alternate index.

If the file is large, applying an alternate-key definition to a file can require considerable processing time. This is because creation of a new alternate index requires that all records in the file be read.

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 a keyed 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 having the specified value for the selected key.

AMP$REWIND
: Positions the file at the beginning of the index for the selected key. The file is positioned to access the record with the lowest value for the selected key.

AMP$SKIP
: Positions the file forward or backward the specified number of records (according to the record order provided by the index for the selected key).

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.

Fetching Access Information After Alternate-Key Selection

An AMP$FETCH_ACCESS_INFORMATION call can return the following items of information. (The call format is in the CYBIL File Management manual.) This list highlights the meaning of each item when returned immediately after a call that specifies an alternate-key value:

duplicate_value_inserted
: Boolean indicating whether the last AMPPUT, AMPPUTREP, AMP$REPLACE, or AMP$APPLY_KEY_DEFINIIONS call detected a duplicate alternate-key value.

  The duplicate_value_inserted item does not identify the duplication. An AMPPUT, AMPPUTREP, or AMP$REPLACE call can detect a duplicate value for any alternate key in the file that allows duplicates. An AMP$APPLY_KEY_DEFINITIONS call can detect a duplicate value for any record in the file.

file_position
: Returns the current file position as described later under File Position Returned.

primary_key
: Primary-key value of the record at the current file position (the next record).

NOTE

The AMP$FETCH_ACCESS_INFORMATION call must specify a pointer to the location where the primary-key value is to be returned. The pointer must be specified in the PRIMARY_KEY field in the array specified by the fetch_items parameter.

---

selected_key_name
: Name of the currently selected key. If the primary key is currently selected, the name $PRIMARY_KEY is returned.

Retrieving Alternate-Index Information

An alternate index is a structure independent from the file data. Thus, a program can fetch information from the alternate index without requiring access to the file data. This section describes the calls that fetch information from the alternate index.

An AMP$GET_KEY_DEFINITIONS call retrieves the definitions of existing alternate keys. Your program could use the definitions returned by AMP$GET_KEY_DEFINITIONS to:

- Determine the attributes of an alternate key
- Define identical or similar alternate keys in another file

For example, you may want to get the alternate-key definitions from an old file to apply to a re-created file.

An AMP$GET_NEXT_PRIMARY_KEY_LIST retrieves primary-key values from the alternate index. The primary-key values are returned in the order the values are stored in the alternate index, beginning at the current position.

Generally, AMP$GET_PRIMARY_KEY_COUNT and AMP$GET_SPACE_USED_FOR_KEY calls prepare for subsequent calls that read or position by alternate key. AMP$GET_PRIMARY_KEY_COUNT counts the number of primary-key values for a range of alternate-key values in the alternate index. AMP$GET_SPACE_USED_FOR_KEY counts the number of alternate-index blocks that contain the specified alternate-key value range.

AMP$GET_PRIMARY_KEY_COUNT gives the program the exact number of primary-key values it would receive if it calls AMP$GET_NEXT_PRIMARY_KEY_LIST for the alternate-key value range. To count the values, AMP$GET_PRIMARY_KEY_COUNT sequentially reads the alternate-index records that contain the information.

AMP$GET_SPACE_USED_FOR_KEY does not actually read the alternate-index records that contain the primary-key values. It just counts the blocks that would contain the records for a given range of alternate-key values. This is much faster. The count returned is generally used to compare with a count returned by another AMP$GET_SPACE_USED_FOR_KEY to determine the shorter primary-key value list.

Program Examples

This section contains CYBIL program examples that perform these functions:

- Create an indexed-sequential file
- Update an indexed-sequential file
- Create and use an alternate key
- Create and delete nested files

```
  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]]];
```

```
?? PUSH (LIST := OFF) ??
{ This deck contains the common procedures listed in appendix E. }
*copyc comproc

*copyc amp$close
*copyc amp$file
*copyc amp$get_next
*copyc amp$open
*copyc amp$put_key
?? POP ??

MODEND create;
```

Assuming the program source text is stored as file $USER.CREATE, the following are the SCL commands required to expand, compile, attach the data files, and execute the program. After the commands is a listing of the statistical messages from the program.

```
/create_source_library base=temporary_library
/scu base=temporary_library
sc/create_deck deck=create modification=original ..
sc../source=$user.create
sc/expand_deck deck=create ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile list=listing
/attach_file $user.original_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.
```

```
  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));
    CASEND;
  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;
   WHILEND;
```

Assuming the program source text is stored on file $USER.UPDATE, the following are the SCL commands required to expand, compile, attach the data file, and execute the program. It is assumed that the indexed-sequential file to be updated is accessible as file INDEXED in the $LOCAL catalog. After the commands is a listing of the statistical messages from the file update program.

```
/create_source_library base=temporary_library
/scu base=temporary_library
sc/create_deck deck=update modification=original ..
sc../source=$user.update
sc/expand_deck deck=update ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile list=listing
/attach_file $user.update_data
/lgo
```

```
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.
```

```
{ 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;
```

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. It is assumed that the indexed-sequential file is accessible as file INDEXED in the $LOCAL catalog.

```
/scu base=$user.my_library
sc../expand_deck deck=(alternate_keys) ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit, write_library=no
/cybil input=compile
/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.
```

Nested File Example

This example is a CYBIL program that first copies the nested-file definitions from one keyed file to another keyed file and then destroys the original nested files.

The program copies the nested-file definitions from file EXISTING_KEYED_FILE to file ANOTHER_KEYED_FILE.

```
MODULE nested_file_module;

  VAR
    lfn1: [STATIC] amt$local_file_name :=
                    'existing_keyed_file',
    lfn2: [STATIC] amt$local_file_name :=
                    'another_keyed_file',
    fid1: amt$file_identifier,
    fid2: amt$file_identifier,
    access_information_ptr: ^amt$access_information,
    definitions_ptr: ^amt$nested_file_definitions,
    nested_file_count: amt$nested_file_count,
    element: amt$nested_file_count;

{ This program copies the nested-file definitions in file
{ EXISTING_KEYED_FILE (LFN1) to file ANOTHER_KEYED_FILE (LFN2).
{ It then deletes all nested files (except $MAIN_FILE) from
{ LFN1.  Any data in the LFN1 nested files (other than in
{ $MAIN_FILE) is discarded.

PROGRAM nested_file_example (VAR program_status: ost$status);

  p#start_report_generation(
    'Start copying of nested-file definitions.');

  amp$open(lfn1, amc$record, NIL, fid1, status);
    p#inspect_status_variable;

{ These statements fetch the number of nested files currently
{ defined in LFN1.

  ALLOCATE access_information_ptr : [1..1];
  access_information_ptr^[1].key:=amc$number_of_nested_files;
  amp$fetch_access_information(fid1, access_information_ptr^,
    status);
    p#inspect_status_variable;
```

```
  p#put_m (TRUE, 'Nested file definition copying is done.');
  p#put_m (TRUE, 'Nested-file deletion now begins.');

{ This loop deletes each nested file in LFN1.  Element 1 in
{ the array is skipped because it contains the definition
{ of nested file $MAIN_FILE which cannot be deleted.

  FOR element := 2 TO nested_file_count DO

    amp$delete_nested_file(fid1,
      definitions_ptr^[element].nested_file_name, status);
    p#inspect_status_variable;

  FOREND;

  amp$close(fid1, status);
    p#inspect_status_variable;

  p#stop_report_generation(
    'Nested-file deletion complete.');

PROCEND nested_file_example;

?? PUSH (LIST := OFF) ??
{ The COMPROC deck contains the common
{ procedures listed in appendix E.
*copyc comproc

*copyc amp$open
*copyc amp$fetch_access_information
*copyc amp$get_nested_file_definitions
*copyc amp$delete_nested_file
*copyc amp$close
*copyc amp$create_nested_file

{ This directive is required to copy the
{ named condition identifier declaration.
*copyc ame$unimplemented_request
?? POP ??

MODEND nested_file_module
```

File Access

You can use a file only if you have access to it. Your access to a file is limited by the permissions you have been granted to the file. You can limit access further by requesting a subset of your permitted access modes when attaching the file. This process is described in the SCL System Interface Usage manual.

The access allowed for a particular instance of open is limited by the access_mode file attribute as specified when the file is opened. The following is a list of the access modes required for each keyed-file interface call.

Call	Access Modes Required
AMP$ABANDON_KEY_DEFINITIONS	Append, shorten, and modify
AMP$APPLY_KEY_DEFINITIONS	Append, shorten, and modify
AMP$CREATE_KEY_DEFINITION	Append, shorten, and modify
AMP$CREATE_NESTED_FILE	Append, shorten, and modify
AMP$DELETE_KEY	Shorten
AMP$DELETE_KEY_DEFINITION	Append, shorten, and modify
AMP$DELETE_NESTED_FILE	Append, shorten, and modify
AMP$GET_KEY	Read (modify required to record statistics)
AMP$GET_KEY_DEFINITIONS	Any access mode
AMP$GET_LOCK_KEYED_RECORD	Read (modify required to record statistics)
AMP$GET_LOCK_NEXT_KEYED_RECORD	Read (modify required to record statistics)
AMP$GET_NESTED_FILE_DEFINITIONS	Any access mode
AMP$GET_NEXT_KEY	Read (modify required to record statistics)
AMP$GET_NEXT_PRIMARY_KEY_LIST	Read
AMP$GET_PRIMARY_KEY_COUNT	Read
AMP$GET_SPACE_USED_FOR_KEY	Read
AMP$LOCK_FILE	Any access mode
AMP$LOCK_KEY	Any access mode
AMP$PUT_KEY	Append (shorten and modify also required if the file has one or more alternate keys)

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 identifying the instance of open (returned by an 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 that has been requested but has not yet been discarded or applied to the nested 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.

- AMP$ABANDON_KEY_DEFINITIONS cannot discard an alternate-key definition that has already been applied to the nested file. To delete an applied alternate-key definition, call AMP$DELETE_KEY_DEFINITION, and then call AMP$APPLY_KEY_DEFINITION to apply the deletion request.

Remarks (Contd)

- If AMC$NO_DUPLICATES_ALLOWED is specified for a new key and the file contains data, AMP$APPLY_KEY_DEFINITIONS returns a nonfatal 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 nonfatal error (such as a duplicate-key value) causes the nonfatal-error limit to be reached and a fatal error to be issued. The fatal error terminates alternate index creation and discards any alternate indexes already built by the call.

  No alternate indexes are created by the terminated AMP$APPLY_KEY_DEFINITIONS procedure; however, it does perform all pending alternate-key deletions.

- Entry of a pause_break_character (usually control-p) is ignored during application of alternate-key definitions.

- Entry of a terminate_break_character (usually control-t) during application of alternate-key definitions returns a prompt to the terminal user, asking for confirmation.

  As described in the prompt, the terminal user should then enter a carriage return or any entry other than RUIN FILE (uppercase or lowercase) to continue the application of alternate-key definitions. Applied alternate-key definitions can be removed without harm to the file after the apply operation has completed.

  A request to ruin the file is not recommended. No file operation can be performed on a ruined file and so no data can be retrieved from the file.

Remarks

- To apply the alternate-key definition specified by an AMP$CREATE_KEY_DEFINITION call to the file, call AMP$APPLY_KEY_DEFINITIONS. 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, that record is ignored.

- Sparse key control is defined by three values:

  Sparse_Key_Control_Position
  Sparse_Key_Control_Characters
  Sparse_Key_Control_Effect

  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-Key Control description in chapter I-1.

- A concatenated key can have up to 64 pieces. The leftmost piece is defined by the key_position and key_length values.

  Each piece concatenated to the first piece is specified by a record in the optional_attributes array containing three fields:

  Concatenated_Key_Position
  Concatenated_Key_Length
  Concatenated_Key_Type

  The pieces are concatenated in the same order as the records that define the pieces in the optional_attributes array.

  The total length of a concatenated key cannot exceed 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).

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
AMC$COLLATE_ TABLE_NAME	COLLATE_TABLE_NAME : pmt$program_name 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 D for more information on collation tables.) If the file is an indexed-sequential file with a collated primary key, the default collation table for the alternate key is the collation table for the primary key. Otherwise, you must specify a collation table for each collated alternate key.
AMC$DUPLICATE_KEYS	DUPLICATE_KEY_CONTROL : amt$duplicate_key_control Indicates how duplicate alternate-key values are handled in the alternate index. AMC$NO_DUPLICATES_ALLOWED No duplicate alternate-key values are allowed in the alternate index. AMC$FIRST_IN_FIRST_OUT Duplicate alternate-key values are ordered according to when the record is written to the file. AMC$ORDERED_BY_PRIMARY_KEY Duplicate alternate-key values are ordered according to primary-key values. Omission causes AMC$NO_ DUPLICATES_ALLOWED to be used.

(Continued)

Table I-3-1. Optional Attribute Record Contents (AMT$OPTIONAL_KEY_ATTRIBUTE) *(Continued)*

Value of SELECTOR Field	Resulting Attribute Record Fields
	AMC$INCLUDE_KEY_VALUE Alternate-key value is included in the alternate index. AMC$EXCLUDE_KEY_VALUE Alternate-key value is not included in the alternate index.
AMC$REPEATING_GROUP	REPEATING_GROUP_LENGTH : amt$max_record_length, 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. REPETITION_CONTROL : amt$repetition_control 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. REPEAT_TO_END_OF_RECORD : boolean TRUE The alternate key repeats until the record ends. (An incomplete key at the end of the record is not used.) FALSE 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.

(Continued)

AMP$CREATE_NESTED_FILE

Purpose Defines a nested file in an existing NOS/VE file.

Format **AMP$CREATE_NESTED_FILE (file_identifier, definition, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

definition: amt$nested_file_definition

Variant record which specifies the nested-file name and its attributes. The record declaration is as follows:

```
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,
    casend,
recend;
```

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Remarks (Contd)

- When creating a direct-access nested file, specify values for the dynamic_home_block_space and loading_factor fields (although the values are not yet used). Specify the default values, FALSE and 0, respectively.

  For the hashing_procedure specification, values are required for two fields (NAME and OBJECT_LIBRARY). Currently, you should always specify OSC$NULL_NAME for the OBJECT_LIBRARY field. To specify the default hashing procedure, specify AMP$SYSTEM_HASHING_PROCEDURE as the NAME field value.

- Creating a nested file does not select the nested file for use. To select a nested file, call AMP$SELECT_NESTED_FILE.

- To remove a nested file, call AMP$DELETE_NESTED_FILE.

- For more information on nested files, see Nested Files in chapter I-1.

- The nested-file example at the end of chapter I-2 demonstrates the use of this call.

Remarks (Contd)

- 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.

- AMP$DELETE_KEY searches for the specified primary-key value only in the nested file currently selected. If it does not find it, it returns the nonfatal condition aae$key_not_found.

- Execution of an AMP$DELETE_KEY call does not change the file position or the currently selected key.

  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.

AMP$DELETE_NESTED_FILE

Purpose Destroys a nested file. It deletes its data, alternate keys, and the nested file definition.

Format **AMP$DELETE_NESTED_FILE (file_identifier, nested_file_name, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

nested_file_name: amt$nested_file_name

Name given the nested file when it was created. It can be specified by an amt$nested_file_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_name
aae$cant_delete_main_nested_f
aae$nested_file_not_found
aae$no_delete_current_nested_f
aae$no_delete_rasp_in_use
aae$no_select_during_keydef
aae$not_enough_permission
aae$system_error_occurred

Remarks

- AMP$DELETE_NESTED_FILE requires append, modify, and shorten access to the file.
- The default nested file $MAIN_FILE cannot be deleted.
- The task must have exclusive access to the nested file to delete it. AMP$DELETE_NESTED_FILE cannot delete a nested file while:
  - Any instance of open has the nested file selected.
  - Any instance of open has any locks that apply to the nested file.

  An attempt to delete a nested file while it is in use returns the nonfatal condition aae$no_delete_rasp_in_use.

AMP$GET_KEY

Purpose

Reads a record from a keyed file using the specified key value.

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 identifying the instance of open (returned by an 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 value of the record to be read. Set to NIL if the key value is an alternate-key value specified in the working storage area.

major_key_length: amt$major_key_length

Length of the major key in bytes. The major key length must be less than or equal to the key length.

If the value is zero, the full key length is used.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

Condition Identifiers

aae$file_at_file_limit
aae$file_is_ruined
aae$key_found_lock_no_wait
aae$key_not_found
aae$major_key_too_long
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$record_longer_than_wsa

Remarks

- To allow for updating of file statistics, you should open the file for both read and modify access.

- If the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is read. The program should either lock the key value before the AMP$GET_KEY call or replace the AMP$GET_KEY call with an AMP$GET_LOCK_KEYED_RECORD call.

  If another instance of open has an Exclusive_Access lock on the primary-key value of the record, AMP$GET_KEY returns the nonfatal condition aae$key_found_lock_no_wait and leaves the file positioned to read the record it found.

  To read about locks, see Keyed-File Sharing in chapter I-2.

- AMP$GET_KEY searches for the specified key value only in the currently selected nested file.

- AMP$GET_KEY can read a record by its primary-key value or by an alternate-key value. The primary key is used unless a preceding AMP$SELECT_KEY call has selected an alternate key.

- If the primary key is selected, the key_location parameter must point to the location of the key value.

- 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.

Remarks (Contd)

- For an indexed-sequential file, 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.)

  When AMP$GET_KEY returns AMC$EOI as the file position, it has not found the requested record and does not return data in the working storage area. It returns AMC$EOI in both of these cases:

  - It is searching for a key value that is greater than or equal to the specified key value and the specified key value is greater than all key values in the file.

  - It is searching for a key value that is greater than the specified key value and the specified key value is the highest value in the file.

Remarks (Contd)

- 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, group_name, sparse_key_control, concatenated_key, and repeating_groups. The first four records are returned for every key definition; the subsequent records are returned only if the attribute was specified for the key definition.

- The attribute order in a key definition may not match the attribute order specified when the alternate key was defined. However, 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.

AMP$GET_LOCK_KEYED_RECORD

Purpose Locks and reads the record having the specified key value.

Format **AMP$GET_LOCK_KEYED_RECORD (file_identifier, working_storage_area, working_storage_length, key_location, major_key_length, key_relation, wait_for_lock, unlock_control, lock_intent, record_length, file_position, wait, status);**

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an 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 value of the record to be read. Set to NIL if the key value is an alternate-key value specified in the working storage area.

major_key_length: amt$major_key_length

Length of the major key in bytes. The major key length must be less than or equal to the key length.

If the major key length is zero, the full key length is used.

This parameter is ignored if the file is a direct-access file and its primary key is currently selected.

Parameters (Contd)

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 at which the file position at completion of the read operation is returned.

AMC$END_OF_KEY_LIST	Positioned at the end of the key list for the specified alternate-key value.
AMC$EOR	Positioned at the end of the record.
AMC$EOI	Positioned at the end-of-information.

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 procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$file_at_file_limit
aae$file_is_ruined
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_not_found
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$major_key_too_long
aae$no_auto_unlock_pc
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$primary_key_locked
aae$record_longer_than_wsa
aae$too_many_keylocks

AMP$GET_LOCK_NEXT_KEYED_RECORD

Purpose Locks and reads the next record.

Format **AMP$GET_LOCK_NEXT_KEYED_RECORD (file_identifier, working_storage_area, working_storage_length, key_location, wait_for_lock, unlock_control, lock_intent, record_length, file_position, wait, status);**

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an 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 key value of the record is returned.

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns a warning condition if the lock is unavailable.

unlock_control: amt$unlock_control

Indicates whether the lock is to be cleared automatically.

AMC$AUTOMATIC	Clear the lock automatically.
AMC$WAIT_FOR_UNLOCK	Keep the lock until it is explicitly unlocked.

Condition Identifiers

aae$bad_resolve_time_limit
aae$cant_da_getn_if_shared
aae$cant_da_getn_after_put
aae$cant_position_beyond_bound
aae$file_at_file_limit
aae$file_boundary_encountered
aae$file_is_ruined
aae$key_already_locked
aae$key_deadlock
aae$key_expired_lock_exists
aae$key_found_lock_no_wait
aae$key_self_deadlock
aae$key_timeout
aae$lock_file_crowded
aae$no_auto_unlock_pc
aae$nonembedded_key_not_given
aae$not_enough_permission
aae$primary_key_locked
aae$record_longer_than_wsa
aae$too_many_keylocks
aae$wsa_not_given

Remarks

- To allow for updating of file statistics, you should open the file for both read and modify access.

- AMP$GET_LOCK_NEXT_KEYED_RECORD performs the same processing as AMP$GET_NEXT_KEY except that it locks the primary-key value of the record before reading the record. See the AMP$GET_NEXT_KEY procedure description for details on how AMP$GET_LOCK_NEXT_KEYED_RECORD finds and reads the record.

- AMP$GET_LOCK_NEXT_KEYED_RECORD requests a lock on the primary-key value of the record to be read. The lock request uses the wait_for_lock, unlock_control, and lock_intent values on the call. For more information on locks, see Keyed-File Sharing in chapter I-2.

- Because a Preserve_Content lock cannot be automatically unlocked, the unlock_control value AMC$AUTOMATIC and the lock_intent value AMC$PRESERVE_CONTENT are not valid on the same call.

- If an alternate key is currently selected, the call requests a lock on the first primary-key value in the key list only.

- If the call terminates abnormally, the primary-key value is left unlocked.

AMP$GET_NESTED_FILE_DEFINITIONS

Purpose Returns the nested-file definitions for the file.

Format **AMP$GET_NESTED_FILE_DEFINITIONS (file_identifier, definitions, nested_file_count, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

definitions: VAR of amt$nested_file_definitions

Array in which the nested-file definitions are returned. Each element is a record of type amt$nested_file_definition as described for the AMP$CREATE_NESTED_FILE procedure.

nested_file_count: VAR of amt$nested_file_count

Variable in which the number of nested files in the file is returned.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$too_little_space
aae$not_enough_permission
aae$system_error_occurred

Remarks

- AMP$GET_NESTED_FILE_DEFINITIONS requires the same access required to open the file.
- The definition of the currently selected nested file is always returned first in the nested_file_definitions array.
- If the nested_file_definitions array is too small for all nested-file definitions in the file, AMP$GET_NESTED_FILE_DEFINITIONS returns the nonfatal condition aae$too_little_space.

  In this case, if sufficient space is available, it returns the definition of the currently selected nested file in the first element of the array, but leaves the rest of the array undefined.

  After receiving the condition aae$too_little_space, a program can use the nested_file_count returned to increase the size of the array to that required for all nested-file definitions and then call AMP$GET_NESTED_FILE_DEFINITIONS again.

AMP$GET_NEXT_KEY

Purpose Reads the next logical record in the keyed file.

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 identifying the instance of open (returned by an 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 value 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, it 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.

Remarks (Contd)

- 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$EOI as the file position; it returns no data and normal status. If the task calls AMP$GET_NEXT_KEY again after AMC$EOI has been returned, the status condition AAE$CANT_POSITION_BEYOND_BOUND occurs.

  For more information on the use of this call with alternate keys, refer to Using Alternate Keys in chapter I-2.

- The key value is returned 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 nonfatal error occurs.

- This call is valid for a direct-access file only when an alternate key is selected or during a sequential pass through the file.

  When the primary key is selected, the call is valid only when the direct-access file has been attached for exclusive access (no share modes allowed) and no update operations intervene in the sequential pass. (The only update operation allowed is the replacement of a record with another record of the same length.)

  If an update operation is performed on the direct-access file and the primary key is selected, the program must rewind the file before beginning a sequential pass of the direct-access file.

Parameters (Contd)

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 list of primary-key values.

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.

Remarks (Contd)

- AMP$GET_NEXT_PRIMARY_KEY_LIST returns primary-key values until it reaches the end of the specified range or until it cannot fit another value into the working storage area. By checking the end_of_primary_key_list value, the program can determine whether all requested values were returned and, if not, call AMP$GET_NEXT_PRIMARY_KEY_LIST again to fetch the rest of the values.

- AMP$GET_NEXT_PRIMARY_KEY_LIST repositions the file as it fetches key values. At completion of the call, the file is positioned at the end of the last key value returned and positioned to continue fetching values at that point if AMP$GET_NEXT_PRIMARY_KEY_LIST is called again.

Parameters (Contd)

major_high_key: amt$major_key_length

A nonzero value indicates that the upperbound alternate-key value is to be located by major key. The nonzero value is the major-key length. A zero value indicates 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-key values 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-key values 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: amt$key_count_limit

Maximum number of primary-key values counted; AMP$GET_PRIMARY_KEY_COUNT stops counting when it reaches this value. If set to zero, all primary-key values are counted.

list_count: VAR of amt$key_count_limit

Integer variable in which the number of primary-key values in the range is returned. If zero is returned, no primary-key values 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 (Contd)

- AMP$GET_PRIMARY_KEY_COUNT returns the value 0 as the list count if it cannot find both the upper_bound and lower_bound alternate-key values in the alternate index.

  For example, if you specify the alternate-key value Z as both the upper_bound and the lower_bound values and the alternate-key value Z is not in the alternate index, the call returns 0 as the list count.

- The list_count_limit value can minimize the processing required for the call. For example, if you call AMP$GET_PRIMARY_KEY_COUNT call to determine whether the number of primary-key values for the alternate-key value Z is 0, 1, or more than 1, you should set the list_count_limit value to 2.

Parameters (Contd)

high_key_relation: amt$key_relation

Indicates where the range ends in relation to the highest value in the range.

AMC$GREATER_KEY

Include the high_key value in the range.

AMC$GREATER_OR_EQUAL_KEY or AMC$EQUAL_KEY

Exclude the high_key value from the range.

data_block_count: VAR of amt$data_block_count

Variable in which the block count is returned. It is returned as an integer from 1 through amt$max_blocks_per_file.

data_block_space: VAR of amt$file_length

Variable in which the combined length of the blocks is returned. (The value is the number of blocks multiplied by the block size.)

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

- The structure of an alternate index is an indexed-sequential structure. One or more index levels are used to find the block containing the alternate-key value. Only the blocks at the lowest level of the search actually contain the alternate-key values and their corresponding primary-key values.

  An AMP$GET_SPACE_USED_FOR_KEY call does not actually find the specified alternate-key values in the alternate index. Rather, it searches the index to determine the number of lowest-level blocks that would contain the specified range of alternate-key values.

  AMP$GET_SPACE_USED_FOR_KEY returns a value even if the specified low_key and high_key values are not in the alternate index.

AMP$LOCK_FILE

Purpose Locks the file.

Format **AMP$LOCK_FILE (file_identifier, wait_for_lock, lock_intent, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns immediately with a warning condition if the lock is unavailable.

lock_intent: amt$lock_intent

Specifies the purpose and effects of the lock.

AMC$EXCLUSIVE_ACCESS	Locked for exclusive access.
AMC$PRESERVE_ACCESS_AND_CONTENT	Locked for possible update requests later.
AMC$PRESERVE_CONTENT	Locked to read records only.

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Condition Identifiers

aae$bad_resolve_time_limit
aae$key_timeout
aae$lock_file_crowded

AMP$LOCK_KEY

Purpose Locks the specified primary-key value.

Format **AMP$LOCK_KEY (file_identifier, key_location, wait_for_lock, unlock_control, lock_intent, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_location: ^cell

Pointer to the primary-key value to be locked.

wait_for_lock: ost$wait_for_lock

Indicates whether the call waits for the lock if it is currently unavailable. The valid values are:

OSC$WAIT_FOR_LOCK	Waits for the lock.
OSC$NOWAIT_FOR_LOCK	Returns immediately with a warning condition if the lock is unavailable.

unlock_control: amt$unlock_control

Indicates whether the lock is automatically cleared.

AMC$AUTOMATIC	The lock is cleared by the next request that reads, updates, or positions the file or requests or clears a lock.
AMC$WAIT_FOR_UNLOCK	The lock is held until it is explicitly cleared.

AMC$AUTOMATIC is not valid if the lock_intent value is AMC$PRESERVE_CONTENT.

Remarks (Contd)

- AMP$LOCK_KEY does not verify that the primary-key value is valid. The validity of the key value is determined by a subsequent call that uses the key value.

- Because a Preserve_Content lock cannot be automatically unlocked, the unlock_control value AMC$AUTOMATIC and the lock_intent value AMC$PRESERVE_CONTENT are not valid on the same call.

- If automatic unlock is not chosen for the key lock, the lock is not cleared until one of these events occurs:

  - An AMP$UNLOCK_KEY call clears the lock.

  - The instance of open is closed.

- For more information, see Keyed-File Sharing in chapter I-2.

Remarks

- An AMP$PUT_KEY call requires that the file be opened for at least append access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate indexes can be updated.

- A lock is not required for an AMP$PUT_KEY call. However, if the file could be shared (more than one concurrent instance of open could exist), the primary-key value of the record should be locked before the record is written. A Preserve_Content_and_Access or Exclusive_Access lock prevents another task from writing a record with the same primary-key value.

  If another instance of open has a lock on the primary-key value, AMP$PUT_KEY returns the nonfatal condition aae$key_found_lock_no_wait.

  To read about file sharing, see Keyed-File Sharing in chapter I-2.

- AMP$PUT_KEY writes the record in the nested file currently selected.

- 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, you should specify the key_location parameter 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.

  A warning message is issued for the first call on which the working_storage_length value differs from the max_record_length value. The warning is given because, although excess data is truncated, insufficient data in the working storage area is not padded. This could mean that garbage has been written as the last part of the fixed-length record.

- Execution of an AMP$PUT_KEY call does not change the key currently selected. It leaves the file positioned at the end of the record it writes.

AMP$PUTREP

Purpose

Either replaces a record if the record is in the keyed file or adds a new record if the record is not in the file.

Format

AMP$PUTREP (file_identifier, working_storage_area, working_ storage_length, key_location, wait, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an 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 value of the new record; specify NIL if the primary key is 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$key_found_lock_no_wait
aae$key_required
aae$nonembedded_key_not_given

Remarks

- An AMP$PUTREP call requires that the file be opened with at least append and shorten access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate indexes can be updated.
- AMP$PUTREP writes or replaces a record in the nested file currently selected.

AMP$REPLACE_KEY

Purpose Replaces an existing record in a keyed file with a new record having the same primary-key value.

Format **AMP$REPLACE_KEY (file_identifier, working_storage_area, working_storage_length, key_location, wait, status);**

Parameters **file_identifier**: amt$file_identifier

File identifier identifying the instance of open (returned by an 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 value of the new record; specify NIL if the primary key is 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$duplicate_alternate_key
aae$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

- An AMP$REPLACE_KEY call requires that the file be opened with at least append and shorten access. If the file has one or more alternate keys, the file must be opened with at least append, shorten, and modify access so that the alternate index can be updated.

AMP$SELECT_KEY

Purpose

Selects the key to be used by subsequent calls that read or position the file.

Format

AMP$SELECT_KEY (file_identifier, key_name, status);

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

key_name: amt$key_name

Name of the key to be used. It can be specified by an amt$key_name variable or by a 31-character string on the call. (The name must be left-justified with blank fill within the string.)

Specify the name $PRIMARY_KEY to switch from an alternate key back to the primary key.

status: VAR of ost$status

Status variable in which the completion status is returned.

Condition Identifiers

aae$altkey_name_not_found
aae$cant_select_key
aae$cant_select_until_applied
aae$no_select_on_pending_delete
aae$not_enough_permission

Remarks

- 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$DELETE_KEY_DEFINITION call has specified the key). If a deletion request is pending for the specified key, AMP$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).

Remarks (Contd)

- AMP$SELECT_NESTED_FILE does not discard the file position, selected key, or locks of previously selected nested files. The instance of open keeps this information for all nested files.

  Thus, a task can sequentially access records on one nested file, select another nested file, reselect the first nested file, and continue the sequential access.

  Similarly, when a task selects an alternate key and then selects another nested file, the alternate key remains selected for the first nested file.

- AMP$SELECT_NESTED_FILE cannot select another nested file if one or more alternate key requests are pending. Call AMP$APPLY_KEY_DEFINITIONS or AMP$ABANDON_KEY_DEFINITIONS to dispose of the pending requests.

- To fetch the name of the currently selected nested file, call AMP$FETCH_ACCESS_INFORMATION to fetch the amc$selected_nested_file item. (AMP$FETCH_ACCESS_INFORMATION is described in the CYBIL File Management manual.)

- For more information on nested files, see Nested Files in chapter I-1.

Parameters (Contd)

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: 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$file_at_file_limit
aae$file_is_ruined
aae$key_not_found
aae$major_key_too_long
aae$no_da_or_sk_start
aae$nonembedded_key_not_given
aae$not_enough_permission

Remarks

- An AMP$START call requires that the file be opened for at least read access.
- AMP$START searches for the specified key value in the nested file currently selected.
- The current file position does not affect AMP$START processing.
- For direct-access files, an AMP$START call is valid only if an alternate key is currently selected. If the primary key is selected, an AMP$START call for a direct-access file returns the nonfatal condition aae$no_da_or_sk_start.

AMP$UNLOCK_FILE

Purpose Clears a file lock.

Format **AMP$UNLOCK_FILE (file_identifier, status);**

Parameters

file_identifier: amt$file_identifier

File identifier identifying the instance of open (returned by an AMP$OPEN call for the file).

status: VAR of ost$status

Status variable in which the procedure returns its completion status.

Remarks

- An AMP$UNLOCK_FILE call clears the file lock for the currently selected nested file only.

  To clear all file locks and all key locks belonging to the instance of open, call AMP$UNLOCK_KEY and specify TRUE for the unlock_all_keys parameter.

- When a lock expires, the task must clear the lock before it can perform any other operations on any nested file in the file.

- For more information, see Keyed-File Sharing in chapter I-2.

Remarks

- AMP$UNLOCK_KEY performs one of two operations depending on the value of the unlock_all_keys parameter:
  - Clears all locks belonging to the instance of open. This includes all file locks and all key locks for all nested files.
  - Clears only the key lock for the primary-key value specified at key_location. The key lock must apply to the currently selected nested file.
- AMP$UNLOCK_KEY cannot clear an individual nested-file lock. To do so, call AMP$UNLOCK_FILE.
- If the call is to unlock all locks, but no locks exists for the instance of open, the call does nothing and returns normal status. However, if the call is to clear a single key lock and the lock does not exist, the call returns the nonfatal condition aae$key_not_previously_locked.
- When a lock expires, the task must clear the lock before it can perform any other operations on any nested file in the file. (A lock can expire only if the lock_expiration_time attribute for the file is not zero.)

  The task is not notified as to which lock has expired. The most direct response to a lock expiration condition is to call AMP$UNLOCK_KEY to clear all locks.

Table I-4-1. Keyed-File Attributes and Access Information Items

FETCH = AMP$FETCH
FETCH_INFO = AMP$FETCH_ACCESS_INFORMATION
FILE = AMP$FILE
GET = AMP$GET_FILE_ATTRIBUTES
OPEN = AMP$OPEN
STORE = AMP$STORE

Attribute or Item	FETCH	FETCH_ INFO	FILE	GET	OPEN	STORE
Access_Mode	X		X	X	X	
Average_Record_ Length	X		X	X	X	
Collate_Table	X					
Collate_Table_ Name	X		X	X	X	
Data_Padding	X		X	X	X	
Duplicate_Value_ Inserted		X				
Embedded_Key	X		X	X	X	
EOI_Byte_Address		X				
Error_Count		X				
Error_Exit_Name	X		X	X	X	
Error_Exit_ Procedure	X					X
Error_Limit	X		X	X	X	X
Error_Status		X				
Estimated_ Record_Count	X		X	X	X	
File_Length				X		
File_Limit	X		X	X	X	
File_Organization	X		X	X	X	
File_Position		X				
Forced_Write	X		X	X	X	
Global_Access_ Mode	X			X		
Global_File_ Name	X			X		

(Continued)

Table I-4-1. Keyed-File Attributes and Access Information Items *(Continued)*

FETCH = AMP$FETCH
FETCH_INFO = AMP$FETCH_ACCESS_INFORMATION
FILE = AMP$FILE
GET = AMP$GET_FILE_ATTRIBUTES
OPEN = AMP$OPEN
STORE = AMP$STORE

Attribute or Item	FETCH	FETCH_ INFO	FILE	GET	OPEN	STORE
Min_Record_ Length	X		X	X	X	
Null_Attribute	X		X	X	X	X
Null_Item		X				
Number_Of_ Nested_Files		X				
Open_Position	X		X	X	X	
Permanent_File	X			X		
Primary_Key		X				
Record_Limit	X		X	X	X	
Record_Type	X		X	X	X	
Records_Per_ Block	X		X	X	X	
Residual_ Skip_Count		X				
Return_Option			X	X	X	
Ring_Attributes	X		X	X	X	
Selected_ Key_Name		X				
Selected_ Nested_File		X				

Default Value The set of access modes defined by the global_access_mode attribute excluding PFC$EXECUTE.

The attribute cannot be changed during the instance of open.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

Table I-4-2. Required Access Modes for Calls

Call	Access Modes Required
AMP$ABANDON_KEY_DEFINITIONS	Append, shorten, and modify
AMP$APPLY_KEY_DEFINITIONS	Append, shorten, and modify
AMP$CREATE_KEY_DEFINITION	Append, shorten, and modify
AMP$CREATE_NESTED_FILE	Append, shorten, and modify
AMP$DELETE_KEY	Shorten
AMP$DELETE_KEY_DEFINITION	Append, shorten, and modify
AMP$DELETE_NESTED_FILE	Append, shorten, and modify
AMP$GET_KEY	Read (modify required to record statistics)
AMP$GET_KEY_DEFINITIONS	Any access mode
AMP$GET_LOCK_KEYED_RECORD	Read (modify required to record statistics; shorten or append required for an Exclusive_Access lock)
AMP$GET_LOCK_NEXT_KEYED_RECORD	Read (modify required to record statistics; shorten or append required for an Exclusive_Access lock)
AMP$GET_NESTED_FILE_DEFINITIONS	Any access mode
AMP$GET_NEXT_KEY	Read (modify required to record statistics)
AMP$GET_NEXT_PRIMARY_KEY_LIST	Read

(Continued)

AVERAGE_RECORD_LENGTH

Meaning

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 value should be the same as the max_record_length value.

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 almost all 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. Although the system uses that value, it does not store the value as the average_record_length value.

Calls

AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

COLLATE_TABLE_NAME

Meaning 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 with the file. The collation table for a file cannot be changed after a new file has been first 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 D, Collation Tables.

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

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.

DUPLICATE_VALUE_INSERTED

Meaning Indicates whether the last AMPPUT, AMPPUTREP, AMP$REPLACE, or AMP$APPLY_KEY_DEFINIIONS call detected a duplicate alternate-key value (access information item).

The duplicate_value_inserted item does not identify the duplication. An AMPPUT, AMPPUTREP, or AMP$REPLACE call can detect a duplicate value for any alternate key in the file that allows duplicates. An AMP$APPLY_KEY_DEFINITIONS call can detect a duplicate value for any record in the file.

Value Boolean value.

TRUE	The last call detected a duplicate alternate-key value.
FALSE	The last call did not detect a duplicate alternate-key value.

Calls AMP$FETCH_ACCESS_INFORMATION.

EMBEDDED_KEY

Meaning Indicates whether the primary key is part of the record data (preserved attribute).

Value Boolean value.

TRUE	The primary key is part of the record data.
FALSE	The primary key is separate from the record data.

Default Value TRUE.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

ERROR_EXIT_NAME

Meaning Name of an error processing procedure (temporary attribute).

The name must be that of a procedure with the XDCL attribute within the program library list 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$PROGRAM_ NAME). (All letters in the name must be uppercase because PMP$LOAD does not convert lowercase letters to uppercase.)

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 the error-exit procedure discussion in the CYBIL File Management Manual.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

ERROR_STATUS

Meaning Completion status returned by the last file interface request for the file (access information item).

Value Integer (type OST$STATUS_CONDITION).

Calls AMP$FETCH_ACCESS_INFORMATION.

ESTIMATED_RECORD_COUNT

Meaning Estimated number of records to be stored in the file (preserved attribute).

The system uses the attribute value to calculate the 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.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

FILE_LENGTH

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

Value Integer from 0 through AMC$FILE_BYTE_LIMIT, 4398046511103 (2^{42}-1) (type AMT$FILE_LENGTH).

Calls AMP$GET_FILE_ATTRIBUTES.

FILE_POSITION

Meaning Current file position (access information item).

Value One of these identifiers that apply to keyed files (type AMT$FILE_POSITION):

AMC$BOI	Beginning-of-information.
AMC$END_OF_KEY_LIST	End of the list of primary keys associated with the same alternate-key value.
AMC$EOR	End of record. (While an alternate key is selected, AMC$EOR indicates that the next record is associated with the same alternate-key value as the current record.)
AMC$EOI	End of information.

Calls AMP$FETCH_ACCESS_INFORMATION.

GLOBAL_ACCESS_MODE

Meaning Indicates the set of valid access modes for the file (returned attribute). (The access modes required for each keyed-file interface call are listed in table I-4-2.)

Value Set of any (including none) of the following constant identifiers (referenced using the set identifier PFTUSAGE_SELECTIONS [ ]):

PFC$READ	Read access.
PFC$SHORTEN	Shorten access.
PFC$APPEND	Append access.
PFC$MODIFY	Modify access.
PFC$EXECUTE	Execute access.

Default Value 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 (read, modify, append, shorten, and execute).

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

GLOBAL_FILE_NAME

Meaning 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 (type OST$BINARY_UNIQUE_NAME).

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

Default Value	The default hashing procedure provided by the system, AMP$SYSTEM_HASHING_PROCEDURE.
Calls	AMP$FETCH, AMP$FILE, AMP$GET_FILE_ ATTRIBUTES, AMP$OPEN.

INDEX_LEVELS

Meaning	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 an indexed-sequential file is created.
Value	1 through 15 (type AMT$INDEX_LEVELS).
Default Value	2.
Calls	AMP$FETCH, AMP$FILE, AMP$GET_FILE_ ATTRIBUTES, AMP$OPEN.

INDEX_PADDING

Meaning	Percentage of space the system is to leave empty in each index block it creates during the first open of an indexed-sequential file (preserved attribute).
Value	0 through 99 (type AMT$INDEX_PADDING).
Default Value	0 (no padding).
Calls	AMP$FETCH, AMP$FILE, AMP$GET_FILE_ ATTRIBUTES, AMP$OPEN.

KEY_TYPE

Meaning Primary-key type (preserved attribute).

For direct-access files, the value specified for the key_type attribute is ignored. The primary-key type for a direct-access file is always uncollated.

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

AMC$UNCOLLATED_KEY
Order key values byte-by-byte according to the ASCII character set sequence (listed in appendix B). Key values can be positive integers or ASCII strings (1 through 255 bytes).

AMC$INTEGER_KEY
Order key values numerically. Key values are positive or negative integers (1 through 8 bytes).

AMC$COLLATED_KEY
Order key values according to a user-specified collation table (see the COLLATE_TABLE_NAME description in this table). Key values can be positive integers or ASCII strings (1 through 255 bytes).

Default Value AMC$UNCOLLATED_KEY.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

Value (Contd)	AMP$GET_NEXT_PRIMARY_KEY_LIST	amc$get_next_primary_key_list
	AMP$GET_PRIMARY_KEY_COUNT	amc$get_primary_key_count
	AMP$GET_SPACE_USED_FOR_KEY	amc$get_space_used_for_key
	AMP$LOCK_FILE	amc$lock_file
	AMP$LOCK_KEY	amc$lock_key
	AMP$OPEN	amc$open_req
	AMP$PUT_KEY	amc$put_key_req
	AMP$PUT_NEXT	amc$put_next_req
	AMP$PUTREP	amc$putrep_req
	AMP$REPLACE_KEY	amc$replace_key_req
	AMP$SELECT_KEY	amc$select_key
	AMP$SELECT_NESTED_FILE	amc$select_nested_file
	AMP$SKIP	amc$skip_req
	AMP$START	amc$start_req
	AMP$STORE	amc$store_req
	AMP$UNLOCK_FILE	amc$unlock_file
	AMP$UNLOCK_KEY	amc$unlock_key

Calls AMP$FETCH_ACCESS_INFORMATION.

MAX_BLOCK_LENGTH

Meaning Length in bytes of each keyed-file block (preserved attribute).

If specified, this value is used only when the keyed file is opened for the first time.

Value Integer from 1 through 16777215 (2^{24}-1). If the value is less than the maximum record length, the system increases it to that value. Then, if needed, it changes the value as follows:

- If the value is less than 2048, it is increased to 2048 (the minimum allocation unit).
- If the value is between 2048 and 65536, but it is not a power of 2, it is increased to the next power of 2 (4096, 8192, 16384, 32768, or 65536).
- If the value is greater than 65536, it is decreased to 65536.

Default Value For an indexed-sequential file, the system calculates an appropriate default value using the average_record_length, estimated_record_count, index_levels, and records_per_block values. For a direct-access file, it calculates the default value using the average_record_length and estimated_record_count values.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

MAX_RECORD_LENGTH

Meaning Maximum length of a file record in bytes (preserved attribute).

Value For keyed files, integer from 1 through 65497.

Default Value For keyed files, no default value is provided; AMP$OPEN returns a fatal error if the maximum record length has not been specified when the file is created.

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

MIN_RECORD_LENGTH

Meaning Minimum record length in bytes (preserved attribute).

Value For keyed files, integer from 0 though 65497, 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 keyed files using embedded keys, the default value is the sum of the key_position and key_length values. Otherwise, the default value is 1.

NOTE

For variable-length records, it is recommended that you explicitly specify the minimum record length. The minimum record length must include:

- The primary-key field
- Any alternate-key fields (or corresponding sparse-key control characters)
- All alternate-key fields for an alternate key defined as a field in a repeating group which repeats a fixed number of times

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

NULL_ATTRIBUTE

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

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN, AMP$STORE.

NULL_ITEM

Meaning Access item identifier that indicates that the content of the attribute record is to be ignored.

Calls AMP$FETCH_ACCESS_INFORMATION.

PERMANENT_FILE

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

Value Boolean value.

TRUE	File is permanent.
FALSE	File is temporary.

Calls AMP$FETCH, AMP$GET_FILE_ATTRIBUTES.

PRIMARY_KEY

Meaning Pointer to a program variable in which the call is to return a primary-key value (access information item).

The primary-key value is for the record at which the preceding AMP$START call positioned the file or for the record read by the preceding AMPGET_NEXT_KEY, AMPGET_LOCK_NEXT_KEY, or AMP$GET_KEY call. This item can be returned only if the preceding call used an alternate key.

Value Cell pointer (type AMT$PRIMARY_KEY).

Calls AMP$FETCH_ACCESS_INFORMATION.

RECORD_LIMIT

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

Value Integer from 1 through AMC$FILE_BYTE_LIMIT ($2^{42}$-1) (type AMT$RECORD_LIMIT).

Default Value AMC$FILE_BYTE_LIMIT ($2^{42}$-1).

Calls AMP$FETCH, AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

RESIDUAL_SKIP_COUNT

Meaning Number of units remaining to be skipped when the skip operation reached a file boundary (access information item). The residual skip count is the difference between the number of skip units requested and the number of units actually skipped.

Value Integer from 0 through AMC$FILE_BYTE_LIMIT (type AMT$RESIDUAL_SKIP_COUNT).

Calls AMP$FETCH_ACCESS_INFORMATION.

RETURN_OPTION

Meaning 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.

NOTE

The task closing the file does not receive notification that the file cannot be detached when the job has another instance of open of the file.

AMC$RETURN_AT_JOB_EXIT	Detach when the job terminates.

Default Value AMC$RETURN_AT_JOB_EXIT.

Calls AMP$FILE, AMP$GET_FILE_ATTRIBUTES, AMP$OPEN.

SELECTED_NESTED_FILE

Meaning Name of the currently selected nested file (access information item). By default, the currently selected nested file is $MAIN_FILE.

Value 31-character string, left-justified, blank-filled (type AMT$NESTED_FILE_NAME). All letters in the name are returned in uppercase.

Calls AMP$FETCH_ACCESS_INFORMATION.

NOTE

To execute a CYBIL program that uses Sort/Merge calls, you must add the following object library to the program library list:

```
$LOCAL.SMF$LIBRARY
```

What Sort/Merge Does

The purpose of sorting is to arrange items in order. The purpose of merging is to combine two or more sets of preordered items. Ordered information makes reports more meaningful and suggests critical relationships. Searches for information are faster with ordered lists.

The purpose of Sort/Merge is to arrange records in the sequence you specify. You describe the files of records that Sort/Merge is to sort and the order in which it is to sort them.

Sort/Merge:

Sorts or merges records from as many as 100 files with one call to Sort/Merge.

Sorts character and noncharacter key types.

Can sort and merge variable-length (V) or fixed-length (F) records.

Can read input records from and write output records to either sequential or indexed-sequential files. (The primary key of each indexed-sequential file must be embedded.)

Can sort according to one of eleven predefined collating sequences, seven numeric formats, or a user-defined collating sequence.

Can sum fields of records having equal keys.

Can use owncode procedures to insert, substitute, modify, or delete records during Sort/Merge processing.

Data Flow

Sort/Merge reads input records from one or more local files or as supplied by an owncode routine. Records to be merged must be presorted. Records to be merged and summed must be pre-sorted and pre-summed.

Sort/Merge writes records to a single output file. The records can be processed by an owncode procedure.

When a university department needs to know which students are majoring in fields within the department, the file can be sorted on the field of study. The same sort can specify the name as a minor key so that records with the same field of study are also sorted in alphabetic order by the name. The file can be sorted by the class code as the major key and by the grade point average in descending numeric order as a minor key. This would produce a list of students sorted by class code with the students having the highest grade point average at the beginning of the list.

Defining a Sort Key

Each sort key to be used by the sort or merge request must be defined by a sort key definition on an SMP$KEY call. A sort key definition includes the following information:

Starting location of the key within the record

Key length

Type of data in the key field

Sort order

Key Length and Position

You define key field length and position by specifying the first byte of the field.

NOTE

When defining a Sort/Merge field, the leftmost byte in a record is counted as number 1.

For example, if you want to specify the name field of the university student record as a sort key, and the name field is the leftmost field in the record, you specify the first byte as 1. If the name field is 20 characters long, you specify the length as 20.

Sort/Merge interprets the integers you specify for key length and position as bit numbers when the key type (discussed later in this chapter) specifies bits; otherwise, byte numbers are assumed. The first bit is numbered 1; the key fields cannot overlap one another and cannot overlap sum fields.

Table II-1-1 summarizes character and noncharacter data types and the associated sort key type.

Table II-1-1. Data in Sort Key Fields

Type	Internal Representation	Data in Field	Type Specified by	Data Ordered According to
Character	ASCII	Alphabetic	Name of a collating sequence	Specified collating sequence
		Numeric	Name of a numeric data format	Numeric value
Noncharacter	Binary value	Numeric	Name of a numeric data format	Numeric value
	Packed decimal numeric	Numeric	Name of a numeric data format	Numeric value

If a sort key field contains any characters that are not meaningful for the key type you specify (an alphabetic character in a field defined as a numeric key, for example), the sort order for that key field in that record is undefined. In the output file, the data for that key field in that record is also undefined. The record is still sorted according to other major sort keys you have specified, unless you have specified an exception file.

The collating sequences and numeric data formats you can specify are discussed in the following paragraphs.

Collating Sequences

A collating sequence determines the precedence given to each character in relation to the other characters. You specify the collating sequence that determines the sort order of character data. (Character data is represented as ASCII character codes.)

Sort/Merge defines six collating sequences: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. (NOS/VE defines five additional collating sequences, and you can define your own collating sequences.)

If you do not specify a collating sequence, ASCII is used. (Sort/Merge sorts fastest when using the ASCII collating sequence.)

The predefined collating sequences are listed in appendix D.

Table II-1-2. Numeric Data Formats

Name	Data Type	Sign	Comments
BINARY	Binary integer	None	The field starts and ends on character boundaries. Data is ordered according to numeric value.
BINARY_BITS	Binary integer	None	The field does not start or end on character boundaries. Data is ordered according to numeric value.
INTEGER	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field starts and ends on character boundaries. Data is ordered according to numeric value.
INTEGER_BITS	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field does not start or end on character boundaries. Data is ordered according to numeric value.
NUMERIC_FS	Leading blanks, numeric characters	- sign for negative values; a + character is not allowed	The field contains leading blanks (leading zeros must be converted to blanks before calling Sort/Merge); if the value is negative, the rightmost leading blank must be converted to a minus sign. If the field contains no leading blanks or does not begin with a negative sign, the value must be positive. This format is equivalent to the FORTRAN I format, or the COBOL picture clause for zero suppressed editing of numeric item. Data is ordered according to numeric value.
NUMERIC_LO	Numeric characters	Leading overpunch	All characters are decimal digits except the leading character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_LS	Numeric characters	Leading separate	All characters are decimal digits except the leading character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_NS	Numeric characters	None	All characters are decimal digits. Data is ordered according to numeric value.
NUMERIC_TO	Numeric characters	Trailing overpunch	All characters are decimal digits except the trailing character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.

Continued

Signed Numeric Data

A floating sign is a negative sign embedded between leading blanks and the numeric characters. A floating sign can also be a negative sign followed by numeric characters. Leading zeros must be converted to blanks. Positive values in this format are not signed. The following examples are valid floating sign formats:

```
    - 1
      1
    - 0
      0
- 1 2 3
1 2 3 4
```

The following examples are invalid floating sign formats:

0 1	Leading zero not allowed
- 0 1	Leading zero not allowed
+ 1 2 3	Positive sign not allowed
	All-blank field not allowed

Diagnostic messages are not issued for invalid floating sign formats or invalid overpunches.

A negative sign overpunch is equivalent to overstriking a digit with a - , which is a punch in row 11. A positive sign overpunch is equivalent to overstriking a digit with a + , which is a punch in row 12.

When a signed overpunch digit is received as input, the digit is punched as indicated in the second column of table II-1-3. When a signed overpunch digit is entered from a terminal or displayed as output, the digit appears as indicated in the third column of table II-1-3. The hexadecimal value is in the fourth column.

Sort Order

Sort/Merge can sort a key in ascending or descending order. If you do not specify a sort order, Sort/Merge sorts the key in ascending order.

When sorting a numeric key in ascending order, Sort/Merge sorts the key values in order from lowest to highest. When sorting a numeric key in descending order, Sort/Merge sorts the key values in order from highest to lowest.

A character key is sorted according to the collating sequence you specify for the key. When sorting a character key in descending order, Sort/Merge sorts the key values in reverse order of the collating sequence you specify.

Specifying the Record Length

Sort/Merge accepts fixed-length (F) or variable-length (V) records. It can sort records up to 65,535 bytes long. The record type and record length are determined by the file attributes specified when the file is created.

The default maximum record length for both fixed-length (F) and variable-length (V) record types is 256 bytes. The default minimum record length for variable-length records is 0 bytes.

If the minimum record length for any Sort/Merge input file is 0, you must include an SMP$KEY call in the Sort/Merge call sequence. If you omit the SMP$KEY call and the minimum record length for any input file is 0, Sort/Merge attempts to use the 0 value (the smallest minimum record length of the input files) as the key length. But Sort/Merge cannot define a key of length 0, so it returns a fatal error.

Sort performance is best when the maximum record length is equal to the longest record to be sorted.

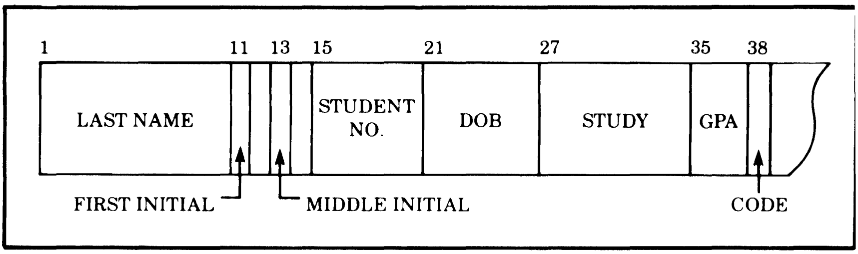
If the SORT or MERGE procedures do not specify any input or output files, Sort/Merge assumes that all records are provided by owncode procedures. In this case, you must specify the record length using either the SMP$OWNCODE_FIXED_RECORD_LENGTH or SMP$OWNCODE_MAX_RECORD_LENGTH procedure.

Sort/Merge determines whether a key or sum field contains valid data when it attempts to use the data. If, when Sort/Merge attempts to compare or sum data from two records, it finds that one record contains invalid data, it then discards the invalid record and attempts to compare or sum the next record. It continues to do so until it finds a record containing valid data. Therefore, in the end cases, where either all records are invalid or the file contains only one record, one record will not be determined as invalid because it cannot be compared or summed with a valid record. So Sort/Merge always outputs at least one record, valid or invalid.

Example Program

The following example CYBIL program sorts a file on three keys.

The file is a file of student records. Each record has this format:



The records are first sorted on the field of study (byte positions 27 through 34 in each record), then on the class code (byte 38), and finally on the student's last name (bytes 1 though 10).

Before a CYBIL program using Sort/Merge is compiled, the source text must be expanded to include the Sort/Merge procedure declarations. See the manual introduction for more information on this process.

Assuming that the source text is on file $USER.SOURCE_TEXT, the following command expand, compile, and execute the example program:

```
/create_source_library result=temporary_library
/source_code_utility base=temporary_library
sc/create_deck deck=sorting source=$user.source_text ..
sc../modification=original
sc/expand_deck deck=sorting ..
sc../alternate_base=($system.cybil.osf$program_interface, ..
sc../$system.common.psf$external_interface_source)
sc/quit write_library=no
/cybil input=compile l=list b=lgo
/attach_file $user.university_students
/lgo
```

Assuming that these records are in file UNIVERSITY_STUDENTS, the program writes the records to the file FIELD_OF_STUDY in this order:

```
REYES     S L 100246031558ANTHRO  3341
MAYER     M I 100991122359ANTHRO  2882
CHARLES   S H 101418032459ANTHRO  2453
MARTIN    R C 100955082157Art     2891
NEECE     M L  99911121358Art     2291
NAKAMURA  S L 101529051260Art     2594
YEH       F L 102005120645Art     2764
BARTLETT  S S 100800100957Art     2735
COCHRAN   G L 100725111857BIO     3011
HOYO      J C 101925103060BIO     3014
.
.
.
KRUTZ     S T 100532010353POLISCI 1981
WALLIN    G E 101056041659POLISCI 3151
WARNES    D V 102116060861POLISCI 2814
WONG      S T 101001012755PSYCH   2152
LANGDON   M A 101754080549PSYCH   2013
LASEUR    P T 100678042256PSYCH   2233
SUGARMAN  B T 100528070457SOC     3501
SMITH     F R 101062120758SOC     2913
DOUGLAS   M L 101325071558UNDEC   2585
OKADA     N A 100103111750UNDEC   2225
```

SMP$BEGIN_SORT_SPECIFICATION

Purpose Signals the beginning of a sort calling sequence of procedure calls.

Format **SMP$BEGIN_SORT_SPECIFICATION (array, status);**

Parameters **array**: VAR of smt$info_array

Result array name; 1 to 31 letters, digits, or the special characters $ # @ _, beginning with a letter.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The SMP$BEGIN_SORT_SPECIFICATION procedure must be the first procedure called for a sort.
- The result array is a 0- through 16-element integer array in which Sort/Merge returns sort statistics and results to your program when the sort is completed. The result array is a single dimensional array.

  You set the first element of the result array to the number of elements (as many as 15) in the result array to receive information. If the first word is set to a value greater than 15 or less than 0, Sort/Merge issues a warning message and changes the value to 15 or 0, respectively.

  The type of result that is returned in each element of the result array is shown in table II-2-1.

SMP$BEGIN_MERGE_SPECIFICATION

Purpose Signals the beginning of a merge calling sequence of procedure calls.

Format **SMP$BEGIN_MERGE_SPECIFICATION (array, status);**

Parameters **array**: VAR of smt$info_array
Result array name; 1 to 31 letters, digits, or the special characters $ # @ _, beginning with a letter.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The SMP$MERGE_SORT_SPECIFICATION procedure must be the first procedure called for a merge.

- The result array is a 0- through 16-element integer array in which Sort/Merge returns merge statistics and results to your program when the merge is completed. The result array is a single dimensional array.

  You set the first element of the result array to the number of elements (as many as 15) in the result array to receive information. If the first word is set to a value greater than 15 or less than 0, Sort/Merge issues a warning message and changes the value to 15 or 0, respectively.

  The type of result that is returned in each element of the result array is shown in table II-2-1.

Remarks (Contd)

- Specifying the file $NULL or an empty FROM file, both without an owncode 1 procedure specified, results in a null sort or merge. A null sort or merge has no records sorted or merged.

- Sort/Merge input files can have either sequential or indexed-sequential file organization and either variable-length (V) or fixed-length (F) record type.

  If an input file is an indexed-sequential file, its primary key must be embedded. If the primary key is nonembedded, Sort/Merge issues a fatal error and terminates.

Remarks (Contd)

- If the output file is an indexed-sequential file, Sort/Merge checks the key_position, key_length, and key_type file attributes.

  - If the major sort key position does not match the key_position attribute value, Sort/Merge issues a fatal error and terminates.

  - If the major sort key length does not match the key_length attribute value, Sort/Merge issues a warning error and changes the major sort key length to match the primary-key length.

  - If the major sort key type does not match the key_type attribute value, Sort/Merge issues a warning error. It also changes the major sort key type if the key_type attribute specifies uncollated or integer keys. (It does not issue a warning or change the key type if the key_type attribute specifies collated keys.)

    - For uncollated keys, the major sort key type is changed to ASCII.

    - For integer keys, the major sort key type is changed to INTEGER.

  To read about indexed-sequential file attributes, see part I of this manual.

Remarks (Contd)

- If the SMP$KEY procedure is not called, the following assumptions are made: the first byte is 1, the key length is the smallest minimum record length of any of the input files, the key type is the ASCII collating sequence, and the sort order is ascending.

- A warning error is issued if a key field contains invalid data. The warning error results in the following actions:

  1. The record is written to the exception records file if an exception records file was specified.

  2. The record is deleted from the sort or merge if an exception file was specified. If an exception records file was not specified, the record remains in the sort or merge, but its place in the sort order is undefined.

  3. A diagnostic message is issued, as controlled by the list options specification.

  4. The sort or merge continues normally.

- If the output (SMP$TO_FILE) file is an indexed-sequential file, the major sort key must be the embedded primary key defined for the output file. For details, see the SMP$TO_FILE procedure description.

SMP$ERROR_FILE

Purpose Specifies the file to which diagnostic messages are written.

Format **SMP$ERROR_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file name of the error file.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- Sort/Merge does not rewind the error file before or after it uses it.
- The file is written in V-type record format. If you specify the file $NULL with the SMP$ERROR_FILE procedure, diagnostic messages are not written.
- If you specify the same file for the listing file and for the error file, each error diagnostic message is written only once, not twice as it would be if the listing file and the error file were different and the messages were written to each file.
- In a batch job, both $LIST and $ERRORS are connected to OUTPUT. With $LIST and $ERRORS connected to the same file each error message is printed twice consecutively. To alleviate this situation you should always set one of the files to a nondefault value, using a value other than OUTPUT.
- If the SMP$ERROR_FILE procedure is not called, errors are written to file $ERRORS.

Table II-2-2. Error Level Specification Using the SMP$ERROR_LEVEL Parameter

Error Level	Errors Reported
'I' or 'i'	Informational, warning, fatal, and catastrophic
'T' or 't'	(This is a nonstandard value and its use is not recommended)
'W' or 'w'	Warning, fatal, and catastrophic
'F' or 'f'	Fatal and catastrophic
'C' or 'c'	Catastrophic
'NONE' or 'none'	None

SMP$EXCEPTION_RECORDS_FILE

Purpose Specifies the file to which invalid records are written.

Format **SMP$EXCEPTION_RECORDS_FILE (file_name, status);**

Parameters **file_name**: string(*)

Local file to which invalid records are written. The file name cannot be the same file name specified by the SMP$TO_FILE procedure. Sort/Merge converts the file name to all uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- If the SMP$EXCEPTION_RECORDS_FILE call specifies the $NULL file, Sort/Merge deletes all exception records. It does not write the exception records to an exception records file or to the output file.

- The records written to the exception records file include:

  - Records containing invalid key or sum field data

  - Records that caused an arithmetic overflow or underflow when their sum fields were summed.

  - Out-of-order merge input records if merge order checking was requested by an SMP$VERIFY call.

  - Records for which the system procedure AMP$PUT_ NEXT returned an error when it attempted to write the record to the output (TO) file.

- The records in the exception file are deleted from the sort or merge. A summary of records written to the exception is printed in the error file named by the SMP$ERROR_FILE procedure call and in the list file.

- If you omit the SMP$EXCEPTION_RECORDS_FILE procedure call, Sort/Merge writes the invalid records to the output file. The invalid records are not written in a defined order.

SMP$LIST_OPTION

Purpose Determines the type of information written to the listing file.

Format **SMP$LIST_OPTION (option, status);**

Parameters **option**: string(*)

Value indicating the listing information requested:

OFF	No additional information is to be written to the listing file.
NONE	Same as the OFF keyword.
S	Although it is a valid keyword, it has no meaning for this CYBIL procedure call. (It is meaningful on the SORT or MERGE command parameter.)
DE	Detailed exception information. A message is written for each occurrence that causes a record to be written to the exception records file. The DE keyword is valid only if you specify an exception records file; otherwise, an informational error message is issued. If you omit the DE keyword, messages are written only once per key, sum fields, or file that causes records to be written to the exception records file.
RS	Record statistics for the records sorted or merged. The statistics are from the result array; a message is written for each element of the array except for the first. Table II-2-1 lists the result array elements.
MS	Merge statistics for the records merged.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$LOAD_COLLATING_TABLE

Purpose Loads a collation table, that is a weight table that defines a collating sequence. The table may be a NOS/VE predefined collation table or a user-defined collation table in an object library.

Format **SMP$LOAD_COLLATING_TABLE (collating_sequence_name, weight_table_name, status);**

Parameters **collating_sequence_name**: string(*)

Name you choose to call the collating sequence produced by the collation table. This name is the name specified in a key field definition. Sort/Merge treats lowercase letters as being equal to uppercase letters.

weight_table_name: string(*)

Name of a predefined collation table or an object library entry point defining a collating sequence. Sort/Merge treats lowercase letters as being equal to uppercase letters.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- A sort or merge specification can include more than one SMP$LOAD_USER_COLLATING_TABLE call.
- The weight table must be loadable by PMP$LOAD.

  For more information on collation tables, see appendix D.
- Your collating sequence name cannot be the name of a predefined collating sequence or the name of a collating sequence you have already defined for the sort or merge.

SMP$OWNCODE_FIXED_RECORD_LENGTH

Purpose

Specifies the number of characters in fixed-length records entering the sort or merge from an owncode routine.

Format

SMP$OWNCODE_FIXED_RECORD_LENGTH (value, status);

Parameters

value: integer

Fixed record length in bytes of all records supplied by any owncode procedure; maximum value is 65,535 bytes.

status: VAR of ost$status

Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The integer you specify is the exact number of bytes in each record; a fatal error results if a record entering the sort from an owncode routine does not have the exact number of bytes.
- If the SMP$OWNCODE_FIXED_RECORD_LENGTH procedure is not called, records entering the sort from an owncode routine can be no longer than the longest allowed input or output record.
- If the sort has no input or output files (records to be sorted are supplied by an owncode routine and sorted records are processed by an owncode routine), you must specify one of the following procedures or else a fatal error results:

  SMP$OWNCODE_FIXED_RECORD_LENGTH
  SMP$OWNCODE_MAX_RECORD_LENGTH

- You cannot call both the SMP$OWNCODE_FIXED_RECORD_LENGTH procedure and the SMP$OWNCODE_MAX_RECORD_LENGTH procedure for the same sort.

SMP$OWNCODE_PROCEDURE_n

Purpose

Specifies an owncode routine to be executed each time a certain event occurs during the sort or merge.

Formats

SMP$OWNCODE_PROCEDURE_1
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_2
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_3
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_4
('procedure_name', status);

SMP$OWNCODE_PROCEDURE_5
('procedure_name', status);

Parameters

procedure_name: string(*)
Owncode procedure name; 1 to 31 uppercase letters, digits, or special characters $ # @ _, beginning with a letter.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The procedure name is the name of the owncode routine. If you enter an owncode routine name in lowercase letters, Sort/Merge will not convert the name to uppercase letters. Use uppercase letters to name a routine.
- Sort/Merge loads the owncode procedures before it begins the sort or merge.
- If the SMP$OWNCODE_PROCEDURE_n procedure is not called, no owncode routine is executed.
- Owncode routines are described in detail in chapter 3.
- You cannot specify both the SMP$OWNCODE_PROCEDURE_5 and SMP$SUM procedure calls for the same sort or merge.
- You cannot specify an owncode 1 or 2 procedure for a merge.

SMP$COLLATING_x

Execution of the SMP$COLLATING_x procedures allow you to define your own collating sequence. A collating sequence specifies the sort or merge order for character data. You must define all 256 characters for the collating sequence or use the SMP$COLLATING_REMAINDER procedure. A collating sequence consists of a series of value steps from low value to high value. Each value step consists of at least one character representation. When a value step contains more than one character, all characters that are named within the step are collated equally.

A sequence of SMP$COLLATING_x procedures defines your collating sequence. Your collating sequence definition starts with the SMP$COLLATING_NAME procedure and ends by any procedure other than SMP$COLLATING_NAME, SMP$COLLATING_CHARACTERS, SMP$COLLATING_REMAINDER, or SMP$COLLATING_ALTER. You can define as many as 100 collating sequences by specifying a separate series of SMP$COLLATING_x procedures for each collating sequence.

SMP$COLLATING_NAME

Purpose

Signals the start of your collating sequence definition and specifies the name of your collating sequence.

Format

SMP$COLLATING_NAME ('name', status);

Parameters

name: string(*)
Your collating sequence name, 1 through 31 characters. The name must be a quoted literal specifying the sequence name.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- Your collating sequence name cannot be the same as the predefined collating sequence names and cannot be the same as a collating sequence you have already defined. Sort/Merge converts your sequence name to uppercase letters.

SMP$COLLATING_ALTER

Purpose Determines whether the characters in the value step defined by the preceding SMP$COLLATING_CHARACTERS call are altered in the output. If altered, all characters in the value step are output as the first character in the value step.

Format **SMP$COLLATING_ALTER ('option', status);**

Parameters **option**: string(*)

YES or Y	Alter characters.
NO or N	Do not alter characters.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$COLLATING_REMAINDER

Purpose Defines the position of the remainder value step in the collating sequence. The remainder value step consists of all characters that have not been included in value steps defined by SMP$COLLATING_CHARACTERS calls.

Format **SMP$COLLATING_REMAINDER ('option', status);**

Parameters **option**: string(*)

YES, Y, NO or N

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

SMP$SUM

Purpose Specifies one or more fields to be summed.

Format **SMP$SUM (first, length, 'stype', rep, status);**

Parameters

first: integer
First byte or bit of the sum field. (Bytes and bits are counted from the left, beginning with 1.)

length: integer
Number of bytes or bits in the sum field.

stype: string(*)
Name of a numeric data format.

rep: integer
Number of times the fields should be repeated to the right; a positive, nonzero integer.

status: VAR of ost$status
Name of the status variable in which Sort/Merge returns the procedure completion status.

Remarks

- The defined sum fields are summed when two records have equal keys. The records with equal keys are combined into one new record.

  The new record contains the equal keys and the summed fields. A data field that is not a key or sum field is written to the new record as a field from one of the old records.

- The location of a sum field is specified as the position as the first bit or byte in the field. Bits and bytes are numbered from the left in the record beginning with 1. The location is a byte position unless the numeric format of the sum field is BINARY_BITS or INTEGER_BITS.

- The maximum size of the BINARY, BINARY_BITS, INTEGER, INTEGER_BITS, PACKED, and PACKED_NS sum fields is one word. The maximum size of NUMERIC_LO, NUMERIC_LS, NUMERIC_TO, NUMERIC_TS, NUMERIC_NS, or NUMERIC_FS sum fields with a nonseparate sign is 17 digits. If the sum fields have a separate sign, the maximum size is 17 digits plus one digit for the sign.

Remarks (Contd)

- A fatal error is issued when a sum field contains invalid data or when an arithmetic overflow or underflow condition occurs as a result of summing two fields. An error due to invalid data leaves the contents of the sum fields undefined; an error due to an arithmetic overflow or underflow leaves valid data in the sum fields, but it may not be the original data.

  A fatal error results in the following actions:

  1. The record or records are written to the exception file if an exception file was specified. (If the error was due to invalid data in a sum field, one record is written; if the error was due to an arithmetic overflow or underflow, both records are written.)

  2. The record or records are deleted from the sort or merge if an exception file was specified. If an exception file was not specified, the record or records remains in the sort or merge, but their place in the sort order is undefined.

  3. A diagnostic message is issued depending on the list options specification.

  4. The sort or merge continues normally.

  If you do not include an SMP$SUM call in the sequence of Sort/Merge calls, records with equal key values are not combined into a single record.

CYBIL owncode procedures that are loaded with the main program and referenced with SMP$OWNCODE_PROCEDURE_n procedure calls must be declared XDCL procedures.

For Sort/Merge to use an object library containing one or more owncode procedures, the object library file must be in the program library list. To add a file to the program library list before executing the CYBIL program, execute a SET_PROGRAM_ATTRIBUTES command.

For detailed information on creating object libraries, see the SCL Object Code Management Usage manual. The example at the end of this chapter stores an owncode procedure in an object library.

Owncode Procedure Parameters

Sort/Merge communicates with an owncode procedure via parameters. The parameters are passed each time Sort/Merge executes the owncode procedure.

Table II-3-1 summarizes the owncode procedures and the parameters passed. Some parameters cannot be omitted; see table II-3-1 for the required parameters.

The parameters passed between Sort/Merge and your owncode procedures are:

VAR return_code: integer
Code altered by an owncode procedure and returned to Sort/Merge

VAR reca: string (*)
Contents of a record

VAR rla: integer
Record length of a record

VAR recb: string(*)
Contents of a second record (owncode 5 procedure only)

VAR rlb: integer
Record length of a second record (owncode 5 procedure only)

The allowed length of records passed to and from an owncode procedure depends on how you have specified the record length, as follows:

- If you have specified the SMP$OWNCODE_FIXED_LENGTH procedure, the number of bytes in the current record must equal the SMP$OWNCODE_FIXED_LENGTH value.

- Otherwise, the maximum record length is determined as the largest value of the following:

  - The maximum_record_length file attribute values of the input or output files

  - The record length value specified by an SMP$OWNCODE_MAX_RECORD_LENGTH procedure call.

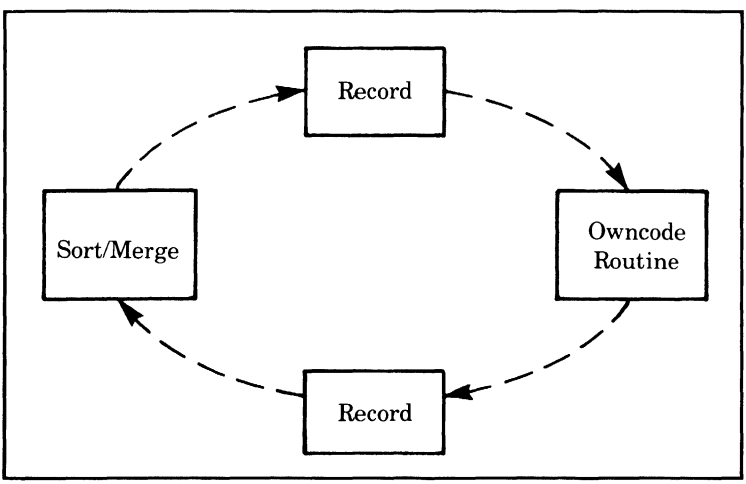
  In this case, the number of bytes in each record can range from 1 through the maximum record length value.

Either the owncode maximum record length or owncode fixed length must be specified if there are no input or output files.

An rla or rlb parameter value that does not correspond to a record specification causes an error.

The contents of the reca, rla, recb, and rlb variables can be altered by an owncode procedure; the routine can pass a different record back to Sort/Merge in reca or recb, and the number of characters in the record can be different.

The record movement from Sort/Merge to an owncode procedure and back to Sort/Merge is shown below.



Input Files Not Specified

If you do not specify any input files (you omit the the SMP$FROM_FILES call from the call sequence), the owncode 1 procedure is executed when Sort/Merge is ready for another record to process. The return_code, reca, and rla parameters are passed to the procedure by Sort/Merge. The return_code is 0, reca is an empty array with enough space for the largest record, and rla is 0.

When control is returned to Sort/Merge from the owncode 1 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The record passed back to Sort/Merge in reca is sorted. The owncode 1 procedure is executed again with reca as an empty array and with rla=0.

2 An additional record is inserted into the sort. The record in reca is entered into the sort, and the owncode 1 procedure is executed again with reca and rla set to the record that just entered the sort.

3 Input is terminated; anything in reca or rla is ignored.

When control is returned to Sort/Merge from the owncode 2 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

0 Signals the end of input.

1 An additional record is inserted into the sort after the last record. The record inserted is the first rla characters in reca, which have been provided by the procedure. The owncode 2 procedure is executed again.

When control is returned to Sort/Merge from the owncode 3 procedure, the return_code value and the associated processing performed by Sort/Merge can be as follows:

1 Owncode 3 is called again.

3 Output is terminated. If an owncode 4 procedure is specified, the procedure is executed; otherwise, the sort or merge is terminated.

Owncode 5: Processing Records With Equal Keys

An SMP$OWNCODE_PROCEDURE_5 procedure call specifies an owncode 5 procedure. Sort/Merge executes the owncode 5 procedure when it encounters two records with equal key values during a sort or merge.

The SMP$OWNCODE_PROCEDURE_5 procedure can be called at any time during the sort or merge whenever Sort/Merge detects duplicate records.

The return_code, reca, rla, recb, and rlb parameters are passed to the procedure by Sort/Merge. The return_code is 0; reca and recb contain the first and second records, respectively, and rla and rlb contain the record lengths in characters of the first and second records, respectively.

After the owncode 5 procedure processes the two records, control is returned to Sort/Merge. Sort/Merge then processes the records according to the return_code value set by the owncode 5 procedure. The return_code value and the associated processing performed by Sort/Merge can be as follows:

0 The first rla characters of reca are accepted as the first record; the first rlb characters of recb are accepted as the second record (the records and record lengths passed back to Sort/Merge can be different from the records and record lengths passed to the owncode procedure).

1 One duplicate record is deleted. The other record is replaced with the first rla characters of reca.

If you call the SMP$RETAIN_ORIGINAL_ORDER procedure in a sort with an owncode 5 procedure, the record that first entered the sort is passed to the owncode 5 procedure as reca; otherwise, either of the two records with equal keys could be passed to the procedure as reca.

The owncode 5 procedure can control the order in which the two records are written to the output file. The record returned to Sort/Merge as reca is written to the output file before the record is returned as recb.

The following command adds $USER.OWN_LIBRARY to the program library list:

```
/set_program_attribute add_library=$user.own_library
```

After executing these commands, a CYBIL program can be executed in which the subroutine OWNCODE can be called from a sequence of Sort/Merge procedure calls such as:

```
smp$begin_sort_specification (iarray, status);
smp$from_file ('university_students', status);
smp$to_file ('field_of_study', status);
smp$key (1, 10, 'ascii', 'a', status);
smp$owncode_procedure_3 ('owncode', status);
smp$end_specification (status);
```

End-Of-Information (EOI)

The point at which data in a file ends. For a keyed file, the EOI file position means that the file is positioned after the record with the highest key value.

Entry Point

A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Exception Records File

As used with the Sort/Merge interface, a file to which invalid records are written before the records are removed from the sort or merge.

External Reference

A reference in one program unit to an entry point in another program unit.

F

F Record Type

Fixed-length records, as defined by the ANSI standard.

Field

A subdivision of a record.

File

A collection of information referenced by a name.

File Attribute

A characteristic of a file. Each file has a set of attributes that define the file structure and processing limitations.

File Cycle

A version of a file. All cycles of a file share the same file entry in a catalog. The file cycle is specified in a file reference by its number or by a special indicator, such as $NEXT.

File Organization

The file attribute that determines the record access method for the file. See Sequential File Organization, Byte-Addressable File Organization, and Keyed File Organization.

File Position

The location in the file at which a subsequent sequential read or write operation would begin.

Index Level Overflow

The condition when a record cannot be written to a file because writing the record would require addition of another index level and the file already has 15 index levels.

Index Record

A record in an index block that associates a key value with a pointer to either a data block or an index block in the next-lower level of the index hierarchy.

Indexed-Sequential File Organization

A keyed-file organization in which records can be read sequentially, ordered by key value, or read randomly by a key value.

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

The key type that orders key values numerically. The key values can be positive or negative integers. Contrast with Collated Key and Uncollated Key.

J

Job

A set of tasks executed for a user name.

K

Key

For Sort/Merge, a key is a part of a record used to determine the position of the record within a sorted sequence of records.

In a keyed file, a key is a value associated with a record as a means of accessing records. It may be a field in the record. See Primary Key and Alternate Key.

Key List

The sequence of primary-key values associated with an alternate-key value in an alternate index. If the alternate key does not allow duplicate values, each key list contains only one value. Otherwise, each key list contains a primary-key value for each record that contains the alternate-key value.

M

Major Key

The leftmost part of a key. The number of bytes to be used is specified as the major key length. A major key can be used to position or read a keyed file.

Major Sort Key

As used with the Sort/Merge interface, a sort key that is the most important and is specified first.

Mass Storage

Disk storage.

Merge

The process of combining two or more presorted files.

Minor Sort Key

As used with the Sort/Merge interface, a sort key that is specified after the major sort key on a SORT or MERGE command or in a procedure call. Minor keys are sorted after the major sort key.

Module

A unit of code. An object module is the unit of object code corresponding to a compilation unit. A load module is a unit of object code stored in an object library.

When using the Debug utility, module refers to a program unit.

N

Nested File

File defined within a keyed file. A nested file is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

Nonembedded Key

A primary key that is not part of the record data. Contrast with Embedded Key.

Null Suppression

Alternate-key attribute indicating that records with null alternate-key values are not included in the alternate index.

Primary Key

The required key in a keyed file. Primary-key value must be unique in the file. See also Alternate Key.

Program-Library List

The list of object libraries searched for modules during program loading. A program-library list search is required to load a collation table module or a Sort/Merge owncode procedure module.

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 mass storage files. Contrast with Sequential Access.

Record

A unit of data that can be read or written by a single I/O request. Also, a set of related data processed as a unit when reading or writing a file.

Repeating Groups

An alternate-key attribute indicating that each data record can contain more than one value for the alternate key.

Rewind

For sequential and byte-addressable files, to position a file at its beginning of information (BOI). For keyed files, to position a file at the record with the lowest key value.

Ring

The level of hardware protection given a file or segment. A file is protected from unauthorized access by tasks executing in higher rings.

Ring_Attributes

A file attribute whose value consists of three ring numbers referenced as r1, r2, and r3. The ring numbers define 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.

Sum Fields

Used with the Sort/Merge interface, a record field containing a numeric value from the corresponding field of another record when the records are summed. The sum of the two values is stored in the new record that is created by the summing.

Summing

Used with the Sort/Merge interface, the process of combining two records having identical key values. The result of the process is a new record containing the original values of the key fields, the summed values of the sum fields, and data from one of the original records in any other record fields.

System Command Language (SCL)

The language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

T

Task

The instance of execution of a program.

U

U Record Type

Records for which the record structure is undefined.

Uncollated Key

A key consisting of 1 to 255 eight-bit characters. These keys are sorted by the magnitude of their binary ASCII code values. Contrast with Collated Key.

V

V Record Type

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

W

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.

Table B-1. ASCII Character Set

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
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

(Continued)

Table B-1. ASCII Character Set *(Continued)*

ASCII Code				
Decimal	**Hexadecimal**	**Octal**	**Graphic or Mnemonic**	**Name or Meaning**
090	5A	132	Z	Uppercase Z
091	5B	133	[	Opening bracket
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

```
amc$max_key_position = 0ffff(16),
amc$max_label_length = osc$maximum_offset;
amc$max_line_number = 6;
amc$max_lines_per_inch = 12,
amc$max_operation = 01ff(16);
amc$max_optional_attributes = 72,
amc$max_page_width = 65535;
amc$max_path_name_size = 256;
amc$max_record_header = 16;
amc$max_records_per_block = 0ffff(16);
amc$max_statement_id_length = 17;
amc$max_tape_mark_count = 40000;
amc$max_user_info = 32;
amc$max_vol_number = 65536;

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$get_next_key_req = 123,
amc$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$clear_space = 7,
amc$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,
```

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$duplicate_value_inserted =
    duplicate_value_inserted: boolean,
  = amc$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$index_levels,
  = amc$null_item =
    ,
  = amc$number_of_nested_files =
    number_of_nested_files: amt$nested_file_count,
  = amc$number_of_volumes =
    number_of_volumes: amt$volume_number,
  = amc$previous_record_address =
    previous_record_address: amt$file_byte_address,
  = amc$previous_record_length =
    previous_record_length: amt$max_record_length,
  = amc$primary_key =
    primary_key: amt$primary_key,
  = amc$residual_skip_count =
    residual_skip_count: amt$residual_skip_count,
  = amc$selected_key_name =
    selected_key_name: amt$selected_key_name,
  = amc$selected_nested_file =
    selected_nested_file: amt$selected_nested_file,
```

```
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;

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);

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$fap_layer_number = 0 .. amc$max_fap_layers;

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,
  = 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$user_info =
  user_info: amt$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,
= 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$error_exit_name =
  error_exit_name: pmt$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,
= 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$key_position =
    key_position: amt$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$file_length = 0 .. amc$file_byte_limit;

amt$file_limit = 0 .. amc$file_byte_limit;

amt$file_lock = (amc$lock_set, amc$already_set);

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$eoi, amc$end_of_key_list);

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;
```

```
= 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$key_type =
    key_type: amt$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$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$label_area_length = 18 .. amc$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;
```

```
amt$nested_file_definitions = array [1 .. * ] of
  amt$nested_file_definition;

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$record_limit = 1 .. amc$file_byte_limit;

amt$record_type = (amc$variable { V } ,
  amc$undefined { U } , amc$ansi_fixed { F } ,
  amc$ansi_spanned { S } , amc$ansi_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$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;
```

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$user_ring_1 = 12 { Reserved for user...O.S. }
{ requests available. };
osc$user_ring_2 = 13;
osc$user_ring_3 = 14 { Reserved for user...O.S. }
{ requests not available. };
osc$user_ring_4 = 15;
```

```
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;
```

PF

Types

```
pft$application_info = string (osc$max_name_size);

pft$permit_options = (pfc$read, pfc$shorten,
  pfc$append, pfc$modify, pfc$execute, pfc$cycle,
  pfc$control);

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;
```


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;
```

Using NOS/VE Predefined Collation Tables

To use one of the NOS/VE predefined collation tables listed at the end of this appendix, you specify the name of the predefined collation table as the collation-table name. Unlike user-defined collation table modules, use of NOS/VE predefined collation tables does not require the addition of an object library to the program-library list.

Sort/Merge Example:

To use the predefined collation table OSV$EBCDIC to define the key type MY_KEY_TYPE, you would include this call in the sequence of Sort/Merge procedure calls:

```
smp$load_collating_table('my_key_type', 'osv$ebcdic', status);
```

Then, to define the first 10 bytes of the record as a key field to be sorted in ascending order using the key type, you would include this Sort/Merge call:

```
smp$key(1, 10, 'my_key_type', 'a', status);
```

Keyed-File Example:

To use the predefined collation table OSV$EBCDIC to order the primary key of a new keyed file, you specify the key type as collated and the collate-table name as OSV$EBCDIC. This is done by initializing two attribute records in the attribute array for an AMP$FILE call before the new keyed file is opened or for the AMP$OPEN call that first opens the new keyed file.

```
[amc$key_type, amc$collated_key],
[amc$collate_table_name, 'OSV$EBCDIC'],
```

Using User-Defined Collation Tables

You can use any collation table stored in an object-library file if you have permission to read the file. To use the collation table, you perform these steps:

1. Specify the collation-table name in the program. (The name must be in the entry-point list of the object library as displayed by a DISPLAY_OBJECT_LIBRARY command.)

2. Add the object library to your program-library list using a SET_PROGRAM_ATTRIBUTE command before executing the program:

   ```
   set_program_attribute add_library=$user.object_library
   ```

Creating a Collation Table

Besides using collation tables created by others, you can also create your own collation tables. The process of using your collation tables was described previously under Using User-Defined Collation Tables.

Creating your own collation table involves these steps:

1. Writing a source code module to initialize the collation table.

2. Compiling the source code module to create the object module.

3. Storing the object module in an object library.

Writing a Module to Initialize a Collation Table

A module to initialize a collation table must perform these steps:

1. Declare a 256-integer array.

2. Store an integer in each element of the array. The integer must be in the range 0 through 255.

The values stored in the array are the collating weights. The collating weight in an array element is the collating weight assigned to the ASCII character corresponding to that element.

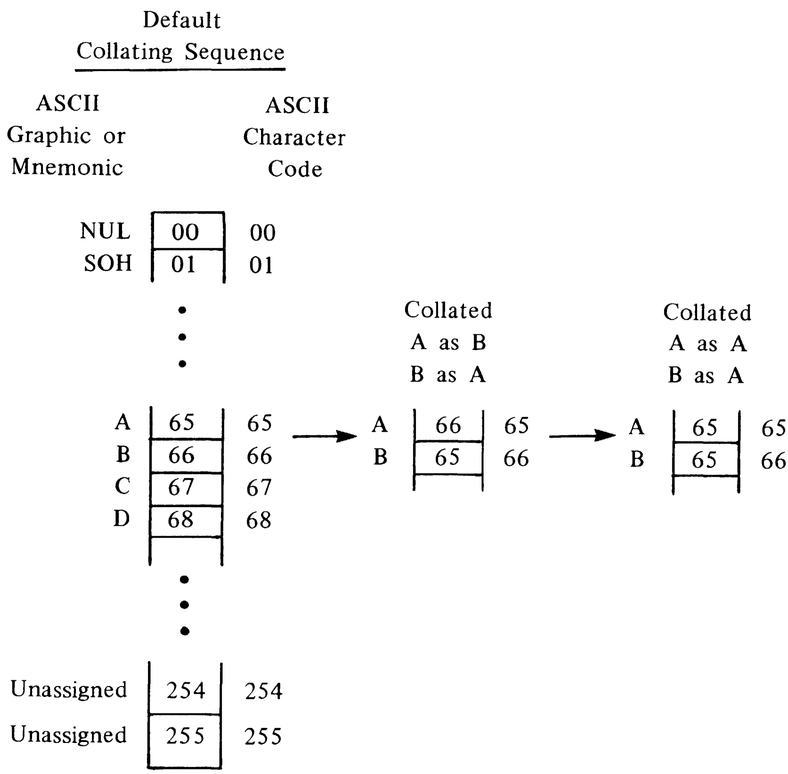
How a Collation Table Works

To determine the correct values with which to initialize the collation table, you must understand how a collation table works.

As shown in figure D-1, each element in the collation table corresponds to an 8-bit character code. The first 128 elements correspond to the 128 characters in the ASCII character set (as listed in appendix B). For example, the element 0 in the table corresponds to the NUL character (character code 00 decimal). Element 65 corresponds to the A character (character code 65 decimal).

Figure D-2 shows how a collation table is initialized for the default ASCII collating sequence. As you can see, the element rank matches the element contents. For example, the element for character NUL (character code 00) contains 0. The element for character A (character code 65) contains 65.

Now, suppose we change two values in the initialized collation table in figure D-2. We change the A element to contain 66 (B) and the B element to contain 65 (A). This collating sequence would order all B characters as A characters and all A characters as B characters. A sort using the collating sequence would sort all B characters before all A characters.



When A and B are ordered; B precedes A.

When A and B are ordered, B is processed as A. (If the value is stored in collated form, B is stored as A.)

Figure D-2. Collation Table Initialized to the Default ASCII Collating Sequence

Sort/Merge Example:

If Sort/Merge used the collation table from figure D-3, it would sort characters as follows:

```
Unordered: 10]garbageGARBAGEgarbage98153;]
Ordered:   10]98153;]aaAAaabBbeEeggGGggrRr
```

Keyed-File Example:

If a keyed file used the collation table from figure D-3, all nonalphabetic key values would be duplicates. Uppercase and lowercase letters would be collated the same, so the key value ABCD would be a duplicate of the key value abcd.

Storing a Module in an Object Library

Source module compilation writes an object module on an object file. You then use the SCL command utility CREATE_OBJECT_LIBRARY to create an object library containing the module. (The CREATE_OBJECT_LIBRARY utility is described in detail in the SCL Object Code Management manual.)

For this example, assume that you have written a CYBIL module (such as the one in figure D-3) to initialize a collation table and that your source text is in file $USER.SOURCE. The following commands compile the program and then store the module on file $USER.COLLATION_LIBRARY

```
/cybil input=$user.source binary_object=object_file ..
../list=list_file
/create_object_library
COL/add_module library=object_file
COL/generate_library library=$user.collation_library
COL/quit
/
```

Sort/Merge uses predefined collation tables for its predefined collating sequences as follows:

Key Type	Predefined Collation Table
ASCII6	OSV$ASCII6_FOLDED
COBOL6	OSV$COBOL6_FOLDED
DISPLAY	OSV$DISPLAY64_FOLDED
EBCDIC	OSV$EBCDIC
EBCDIC6	OSV$EBCDIC6_FOLDED

The Sort/Merge key type ASCII uses the default ASCII collating sequence; it does not use any of the predefined collating sequences listed in this appendix.

Table D-1. OSV$ASCII6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
47	4F,6F	O,o	Uppercase O, lowercase o
48	50,70	P,p	Uppercase P, lowercase p
49	51,71	Q,q	Uppercase Q, lowercase q
50	52,72	R,r	Uppercase R, lowercase r
51	53,73	S,s	Uppercase S, lowercase s
52	54,74	T,t	Uppercase T, lowercase t
53	55,75	U,u	Uppercase U, lowercase u
54	56,76	V,v	Uppercase V, lowercase v
55	57,77	W,w	Uppercase W, lowercase w
56	58,78	X,x	Uppercase X, lowercase x
57	59,79	Y,y	Uppercase Y, lowercase y
58	5A,7A	Z,z	Uppercase Z, lowercase z
59	5B,7B	[,{	Opening bracket, opening brace
60	5C,7C	\ , |	Reverse slant, vertical line
61	5D,7D	] , }	Closing bracket, closing brace
62	5E,7E	^ ,~	Circumflex, tilde
63	5F	_	Underline

Table D-2. OSV$ASCII6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

Table D-3. OSV$COBOL6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
47	55,75	U,u	Uppercase U, lowercase u
48	56,76	V,v	Uppercase V, lowercase v
49	57,77	W,w	Uppercase W, lowercase w
50	58,78	X,x	Uppercase X, lowercase x
51	59,79	Y,y	Uppercase Y, lowercase y
52	5A,7A	Z,z	Uppercase Z, lowercase z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table D-4. OSV$COBOL6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

Table D-5. OSV$DISPLAY63_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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,7B	[, {	Opening bracket, opening brace
49	5D,7D	] , }	Closing bracket, closing brace
50	3A	:	Colon
51	22	"	Quotation marks
52	5F	_	Underline
53	21	!	Exclamation point
54	26	&	Ampersand
55	27	'	Apostrophe
56	3F	?	Question mark
57	3C	<	Less than
58	3E	>	Greater than
59	40,60	@,`	Commercial at, grave accent
60	5C,7C	\ , |	Reverse slant, vertical line
61	5E,7E	^ ,~	Circumflex, tilde
62	3B	;	Semicolon

Table D-6. OSV$DISPLAY63_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

Table D-7. OSV$DISPLAY64_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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,7B	[, {	Opening bracket, opening brace
50	5D,7D	] , }	Closing bracket, closing brace
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@ , `	Commercial at, grave accent
61	5C,7C	\ , |	Reverse slant, vertical line
62	5E,7E	^ , ~	Circumflex, tilde
63	3B	;	Semicolon

Table D-8. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
044	8C	—	Unassigned
045	05	ENQ	Enquiry
046	06	ACK	Acknowledge
047	07	BEL	Bell
048	90	—	Unassigned
049	91	—	Unassigned
050	16	SYN	Synchronous idle
051	93	—	Unassigned
052	94	—	Unassigned
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

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
132	64	d	Lowercase d
133	65	e	Lowercase e
134	66	f	Lowercase f
135	67	g	Lowercase g
136	68	h	Lowercase h
137	69	i	Lowercase i
138	C4	—	Unassigned
139	C5	—	Unassigned
140	C6	—	Unassigned
141	C7	—	Unassigned
142	C8	—	Unassigned
143	C9	—	Unassigned
144	CA	—	Unassigned
145	6A	j	Lowercase j
146	6B	k	Lowercase k
147	6C	l	Lowercase l
148	6D	m	Lowercase m
149	6E	n	Lowercase n
150	6F	o	Lowercase o
151	70	p	Lowercase p
152	71	q	Lowercase q
153	72	r	Lowercase r
154	CB	—	Unassigned
155	CC	—	Unassigned
156	CD	—	Unassigned
157	CE	—	Unassigned
158	CF	—	Unassigned
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

(Continued)

Table D-9. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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 D-10. OSV$EBCDIC6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
48	55,75	U,u	Uppercase U, lowercase u
49	56,76	V,v	Uppercase V, lowercase v
50	57,77	W,w	Uppercase W, lowercase w
51	58,78	X,x	Uppercase X, lowercase x
52	59,79	Y,y	Uppercase Y, lowercase y
53	5A,7A	Z,z	Uppercase Z, lowercase z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table D-11. OSV$EBCDIC6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

```
{ ------------------------------------------------------------------- }
{ 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 the other procedures. }
{ -------------------------------------------------------------------- }

  PROCEDURE p#start_report_generation (startup_message : STRING ( * )) ;

    VAR
      file_access_selection_p : ^ ARRAY [1 .. *] OF AMT$ACCESS_SELECTION ;
                                                  { used by AMP$OPEN_FILE }

    error_count := -0 ;                      { initialize error counting }

    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 ;
                                        { must be positioned for append access }
    AMP$OPEN (report_file_name, AMC$RECORD, file_access_selection_p,
              report_file_identifier, status) ;
    FREE file_access_selection_p ;

    text_index := 1 ;
    text_line(text_index, 1) := '0' ;           { carriage control character }
    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 ,                        { formatting area length }
      paper  : STRING ( 75 ) ;                  { formatting area        }

    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) ;

    AMP$CLOSE (report_file_identifier, status) ;

  PROCEND p#stop_report_generation ;
```

```
      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 ;                       { reset index  }
        text_line(1, line_length) := ' ' ;      { blank filler }
        text_index := text_index + 1 ;          { leave column 1 as carriage }
                                                { control character          }
        p#put_m (new_line_flag,
                 message_string(string_position_locator + 1, *)) ;
      IFEND ;
    IFEND ;

  PROCEND p#put_m ;


{ ---------------------------------------------------------------------------}
{ This routine looks at the global status variable.  If something has gone   }
{ wrong, then 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 has become too large.}
{ ---------------------------------------------------------------------------}

  PROCEDURE [INLINE] p#inspect_status_variable ;

    IF NOT status.normal
      THEN
      error_count := error_count + 1 ;        { increment error counter     }
      IF error_count < 333
        THEN
        p#display_status_variable ;           { issue the message           }
      ELSEIF error_count = 333
        THEN
        p#put_m (TRUE,
             'Error_Count = 333.  Further message reporting is turned off.') ;
      IFEND ;
    IFEND ;

  PROCEND p#inspect_status_variable ;
```

F

J

K

L

M

T

U

V

W

Z

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MN.

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

Publications and Graphics Division
Mail Stop: SVL104
P.O. Box 3492
Sunnyvale, California 94088-3492

FOLD FOLD

Comments (continued from other side)