CYBIL for NOS/VE System Interface

Usage

60464115

Index of Program Interface Procedures

CLP$COLLECT_COMMANDS 9-65
CLP$CONVERT_INTEGER_TO_RJSTRING 8-13
CLP$CONVERT_INTEGER_TO_STRING 8-11
CLP$CONVERT_STRING_TO_FILE 8-17
CLP$CONVERT_STRING_TO_INTEGER 8-15
CLP$CONVERT_STRING_TO_NAME 8-16
CLP$CONVERT_VALUE_TO_STRING 8-18
CLP$CREATE_VARIABLE 8-3
CLP$DELETE_VARIABLE 8-5
CLP$END_SCAN_COMMAND_FILE 9-38
CLP$GET_COMMAND_ORIGIN 9-66
CLP$GET_DATA_LINE 9-67
CLP$GET_PARAMETER 9-18
CLP$GET_PARAMETER_LIST 9-19
CLP$GET_PATH_DESCRIPTION 9-21
CLP$GET_SET_COUNT 9-14
CLP$GET_VALUE 9-17
CLP$GET_VALUE_COUNT 9-15
CLP$GET_WORKING_CATALOG 9-24
CLP$POP_PARAMETERS 9-28
CLP$POP_UTILITY 9-35
CLP$PUSH_PARAMETERS 9-27
CLP$PUSH_UTILITY 9-31
CLP$READ_VARIABLE 8-6
CLP$SCAN_ARGUMENT_LIST 9-57
CLP$SCAN_COMMAND_FILE 9-37
CLP$SCAN_COMMAND_LINE 9-68
CLP$SCAN_EXPRESSION 9-63
CLP$SCAN_PARAMETER_LIST 9-12
CLP$SCAN_PROC_DECLARATION 9-71
CLP$SCAN_TOKEN 9-61
CLP$SET_WORKING_CATALOG 9-25
CLP$TEST_PARAMETER 9-13
CLP$TEST_RANGE 9-16
CLP$WRITE_VARIABLE 8-8
OFP$DISPLAY_STATUS_MESSAGE 2-31
OFP$RECEIVE_FROM_OPERATOR 2-33
OFP$SEND_TO_OPERATOR 2-32
OSP$APPEND_STATUS_INTEGER 6-5
OSP$APPEND_STATUS_PARAMETER 6-4
OSP$AWAIT_ACTIVITY_COMPLETION 4-2
OSP$FORMAT_MESSAGE 6-12
OSP$GET_MESSAGE_LEVEL 6-14
OSP$GET_STATUS_SEVERITY 6-9

(Continued on inside back cover)

CYBIL for NOS/VE System Interface

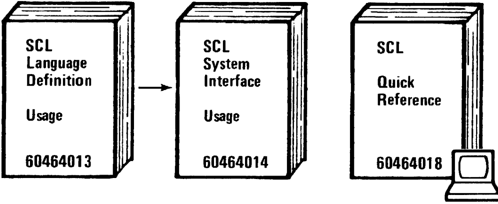
Usage

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.

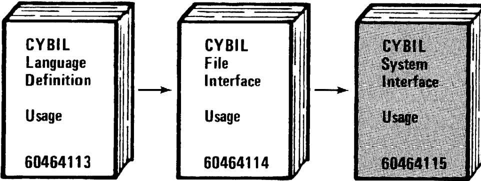
Publication Number 60464115

Related Manuals

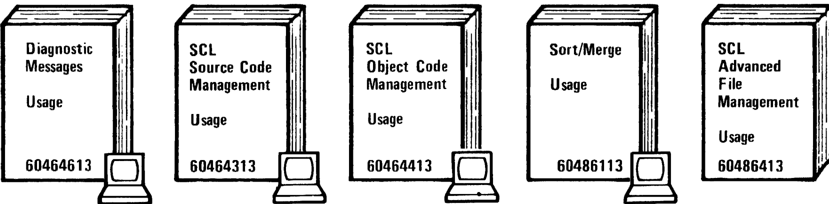
Background (Access as Needed):



CYBIL Manual Set:



Additional References:



→ indicates a reading sequence.

means available online.

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

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 7

Audience 7
Organization 8
Conventions 9
Additional Related Manuals 10
Ordering Manuals 10
Submitting Comments 10

How to Use System Interface Calls 1-1

Using System Interface Procedures 1-1
System Naming Convention 1-10
Procedure Call Description Format 1-12

Program Services 2-1

Date and Time Retrieval 2-1
System Information Retrieval 2-12
Job Information Retrieval 2-17
Sense Switch Management 2-25
Job Log Messages 2-28
Operator Messages 2-30

Program Execution 3-1

Program Description 3-1
Task Parameters 3-9
Task Initiation 3-10
Task Dependencies 3-14
Task Termination 3-17

Task Communication 4-1

General Wait 4-1
Job-Local Queues 4-4
Queue Information Retrieval 4-12
Queue Communication Example 4-16

Condition Processing 5-1

System Condition Detection 5-1
Condition Handling 5-4

Message Generation 6-1

Status Record Generation 6-1
Status Severity Check 6-7
Message Formatting 6-10

Interstate Communications 7-1

Creating a NOS Job 7-1
Starting a NOS Job 7-6
Communication Between the Task and Job 7-7
NOS Job Communication with the NOS/VE Task 7-10
Interstate Communication Example 7-17

Command Language Services 8-1

Command Language Variables 8-1
String Conversion Procedures 8-10

Command Language Processing 9-1

Command Processor 9-1
Command Utility 9-29
Utility Functions 9-52
Token Scanning 9-58
Expression Evaluation 9-62
Command File Input 9-64
Scanning Declarations 9-69

Glossary A-1

ASCII Character Set B-1

Constant and Type Declarations C-1

AV C-1
CL C-1
IF C-12
JM C-13
MM C-14
OF C-15
OS C-16
PF C-23
PM C-27

Stack Frame Save Area D-1

Index Index-1

About This Manual

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

NOS/VE provides a 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.

Organization

This manual is organized as follows:

Chapter 1 is an introduction to the use of system-supplied CYBIL calls. You should read this chapter first.

Each of the subsequent chapters describes a particular function. You can read these chapters in any order. The primary functions described in these chapters are program management, interstate communication, and command language processing.

Appendixes provide a glossary, an ASCII character set table, the constant and type declarations used by procedures described in the manual, and a description of the stack frame area.

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

- The CYBIL Language Definition manual, which defines the CYBIL language in detail.
- The CYBIL File Interface manual, which describes the CYBIL procedures that NOS/VE supplies for file I/O.

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.

Additional Related Manuals

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

Ordering Manuals

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

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

Submitting Comments

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

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

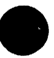
Please indicate whether you would like a written response.

How to Use System Interface Calls 1

Using System Interface Procedures 1-1
Copying Procedure Declaration Decks 1-3
Expanding a Source Program 1-4
Calling a System Interface Procedure 1-6
Parameter List 1-6
Checking the Completion Status 1-8
Exception Condition Information 1-9

System Naming Convention 1-10

Procedure Call Description Format 1-12



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.

```
MODULE example1;

{     SCU directive to copy the procedure declaration. }

*copyc PMP$GET_TIME

  PROGRAM time_retrieve;

{     Statements declaring parameter variables. }

  VAR
    time_returned: OST$TIME,
    status: OST$STATUS;

{     Procedure call statement. }

    PMP$GET_TIME (OSC$AMPM_TIME, time_returned, status);

{     Status record check. }

    IF NOT status.NORMAL THEN
      RETURN;
    IFEND;

    PROCEND time_retrieve;
MODEND example1;
```

Figure 1-1. System Interface Program Example

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

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.

Expanding a Source Program

A CYBIL source program that calls system interface procedures must be expanded to include the XREF procedure declarations specified on any *COPYC directives. To be expanded, the program must exist as one or more decks on an SCU library. The contents of a file containing a source program are transferred onto a deck when the CREATE_DECK subcommand is used within an SCU session. An example of how this is accomplished is listed in figure 1-2.

```
1. /create_source_library
2. /scu base=result result=$user.my_library
3. sc/create_deck deck=my_program modification=mod0 ..
   sc../source=source_file
4. sc/expand_deck deck=my_program ..
   sc../alternate_base=$system.cybil.osf$program_interface
5. sc/quit write_library=true
6. /cybil i=compile l=listing
```

Figure 1-2. Source Text Preparation Example

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

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.

Calling a System Interface Procedure

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

```
procedure_name (parameter_list);
```

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

Parameter List

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

NOTE

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

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

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.

Checking the Completion Status

The last parameter on a system interface procedure call must be a status variable (type OST$STATUS). Unlike the status parameter on SCL commands, the status parameter on system interface calls is required, not optional.

NOS/VE initializes the status variable to normal status when it begins executing the procedure and returns the procedure status in the variable when the procedure is completed.

Your program should check the completion status returned immediately after the procedure call. If the first field of the status record, NORMAL, is TRUE, the procedure completed normally. If the NORMAL field is not TRUE, the procedure completed abnormally.

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

```
pmp$get_time (osc$ampm_time, time_returned, stat);
IF NOT stat.NORMAL THEN
   RETURN;
IFEND;
```

An example of a method to investigate an abnormal status record is found in chapter 6, figure 6-1.

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.

System Naming Convention

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

idx$name

Field	Description
id	Two characters identifying the product that uses the identifier. Table 1-1 lists the identifiers used in this manual.
x	Character indicating the CYBIL element type identified as follows:

x	Description
c	Constant.
d	Declaration.
e	Error condition.
p	Procedure.
s	Memory section.
t	Type.
v	Variable.

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

NOTE

To avoid matching a Control Data-defined identifier, you should avoid using the $ character in any identifier you define.

Field	Description
name	A string of characters uniquely identifying the request.

For example, the identifier PMP$GET_TIME follows the system naming convention. Its process id is PM for program management. The P following the process id indicates it is a procedure name. The string GET_TIME describes the purpose of the procedure.

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

Procedure Call Description Format

Each of the remaining chapters of this manual describe a group of system interface procedures. Within a chapter are individual procedure call descriptions. Each procedure description uses the same format and has the following subheadings.

Purpose	Brief statement describing the purpose of the procedure.
Format	Format of the procedure call. It shows the parameters in positional order.
Parameters	Descriptions of the parameters in the preceding format including the parameter's type.
Condition Identifiers	List of condition identifiers returned by the procedure that are most likely to be of interest when using the procedure. The error messages for the condition identifiers are listed in the Diagnostic Messages manual.
Remarks	If present, additional information about procedure processing.

Each parameter description states the parameter's purpose within the CYBIL call and the valid values for the parameter. The parameter's required CYBIL type is also listed. Appendix C lists the CYBIL types alphabetically.

If the parameter type is a CYBIL set type, the parameter description lists all possible identifiers in the set and their meanings.

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

Program Services 2

Date and Time Retrieval 2-1
PMP$GET_DATE 2-2
PMP$GET_TIME 2-3
PMP$GET_LEGIBLE_DATE_TIME 2-4
PMP$GET_COMPACT_DATE_TIME 2-6
PMP$COMPUTE_DATE_TIME 2-7
PMP$FORMAT_COMPACT_DATE 2-8
PMP$FORMAT_COMPACT_TIME 2-9
Date and Time Retrieval Example 2-10

System Information Retrieval 2-12
PMP$GET_MICROSECOND_CLOCK 2-13
PMP$GENERATE_UNIQUE_NAME 2-14
PMP$GET_OS_VERSION 2-15
PMP$GET_PROCESSOR_ATTRIBUTES 2-16

Job Information Retrieval 2-17
PMP$GET_ACCOUNT_PROJECT 2-18
PMP$GET_JOB_MODE 2-19
PMP$GET_JOB_NAMES 2-20
PMP$GET_SRUS 2-21
PMP$GET_TASK_CP_TIME 2-22
PMP$GET_TASK_ID 2-23
PMP$GET_USER_IDENTIFICATION 2-24

Sense Switch Management 2-25
PMP$MANAGE_SENSE_SWITCHES 2-26
Sense Switch Example 2-27

Job Log Messages 2-28
PMP$LOG 2-29

Operator Messages 2-30
OFP$DISPLAY_STATUS_MESSAGE 2-31
OFP$SEND_TO_OPERATOR 2-32
OFP$RECEIVE_FROM_OPERATOR 2-33

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_DATE

Purpose Returns the current date in legible format.

Format **PMP$GET_DATE (format, date, status)**

Parameters **format**: ost$date_formats;

Format in which date is returned.

OSC$MONTH_DATE

Format month day, year.
For example, November 13, 1982.

OSC$MDY_DATE

Format month/day/year.
For example, 11/13/82.

OSC$DMY_DATE

Format day/month/year.
For example, 13/11/82.

OSC$ISO_DATE

Format year-month-day.
For example, 1982-11-13.

OSC$ORDINAL_DATE

Format yearday.
For example, 1982317.

OSC$DEFAULT_DATE

Default format selected during NOS/VE installation.

date: VAR of ost$date;

Date returned.

status: VAR of ost$status;

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

Condition Identifier pme$invalid_date_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

PMP$GET_LEGIBLE_DATE_TIME

Purpose Returns the current date and time in legible format.

Format **PMP$GET_LEGIBLE_DATE_TIME (date_format, date, time_format, time, status)**

Parameters **date_format** : ost$date_formats;

Format in which date is returned.

OSC$MONTH_DATE

Format month day, year.
For example, November 13, 1982.

OSC$MDY_DATE

Format month/day/year.
For example, 11/13/82.

OSC$DMY_DATE

Format day/month/year.
For example, 13/11/82.

OSC$ISO_DATE

Format year-month-day.
For example, 1982-11-13.

OSC$ORDINAL_DATE

Format yearday.
For example, 1982317.

OSC$DEFAULT_DATE

Default format selected during NOS/VE installation.

date : VAR of ost$date;

Date returned.

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$GET_COMPACT_DATE_TIME

Purpose Returns the current date and time in a compact format.

Format **PMP$GET_COMPACT_DATE_TIME (date_time, status)**

Parameters **date_time:** VAR of ost$date_time;

Date and time returned.

status: VAR of ost$status;

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

Condition Identifier pme$computed_year_out_of_range

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_DATE

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

Format **PMP$FORMAT_COMPACT_DATE (date_time, format, date, status)**

Parameters **date_time**: ost$date_time;

Date and time returned by the PMP$GET_COMPACT_DATE_TIME procedure.

format: ost$date_formats;

Legible date format.

OSC$MONTH_DATE

Format month day, year.
For example, November 13, 1982.

OSC$MDY_DATE

Format month/day/year.
For example, 11/13/82.

OSC$DMY_DATE

Format day/month/year.
For example, 13/11/82.

OSC$ISO_DATE

Format year-month-day.
For example, 1982-11-13.

OSC$ORDINAL_DATE

Format yearday.
For example, 1982317.

OSC$DEFAULT_DATE

Default format selected during NOS/VE installation.

date: VAR of ost$date;

Date in legible format.

status: VAR of ost$status;

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

Condition Identifiers

pme$invalid_date_format
pme$invalid_day
pme$invalid_month
pme$invalid_year

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

Date and Time Retrieval Example

The following source text makes calls to CYBIL procedures to get the current date and time, compute a new date, and convert the date and time to a legible format.

```
MODULE date_time_example;

*copyc pmp$get_compact_date_time
*copyc pmp$compute_date_time
*copyc pmp$format_compact_date
*copyc pmp$format_compact_time

PROGRAM month_ahead (VAR print_date: ost$date;
  VAR print_time: ost$time);

  VAR
    base_date_time: ost$date_time,
    status: ost$status,
    increment: pmt$time_increment,
    new_date_time: ost$date_time;
    print_date: ost$date;
    print_time: ost$time;

/time_example_block/
  BEGIN
    pmp$get_compact_date_time(base_date_time, status);
    IF NOT status.normal THEN
      EXIT /time_example_block/;
    IFEND;
```

```
{ 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;
```

System Information Retrieval

The following procedures return miscellaneous items of information.

PMP$GET_MICROSECOND_CLOCK

Returns the current value of the microsecond clock (64-bit integer).

PMP$GENERATE_UNIQUE_NAME

Returns a valid name unique within all NOS/VE systems.

PMP$GET_OS_VERSION

Returns the operating system name and version number.

PMP$GET_PROCESSOR_ATTRIBUTES

Returns central processor attributes. Central processor attributes include the system model, serial number, and page size.

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$GENERATE_UNIQUE_NAME

Purpose Generates a name that is unique to all NOS/VE systems.

Format **PMP$GENERATE_UNIQUE_NAME (name, status)**

Parameters **name**: VAR of ost$unique_name;

Name record generated. The record is a tagless variant as shown in the OST$UNIQUE_NAME type declaration in appendix C.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks This procedure is useful when the new name need not be meaningful, but must not duplicate an existing name.

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.

PMP$GET_PROCESSOR_ATTRIBUTES

Purpose Returns central processor attributes.

Format **PMP$GET_PROCESSOR_ATTRIBUTES (attributes, status)**

Parameters **attributes**: VAR of pmt$processor_attributes;

Central processor attributes.

Field	Content
model_number	Model number (type PMT$CPU_MODEL_NUMBER). PMC$CPU_MODEL_P1. PMC$CPU_MODEL_P2. PMC$CPU_MODEL_P3. PMC$CPU_MODEL_P4
serial_number	Number uniquely identifying the processor (type PMT$CPU_SERIAL_NUMBER).
page_size	Number of bytes in a memory page (type OST$PAGE_SIZE).

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_ACCOUNT_PROJECT

Purpose

Returns the account name and project name to which task resource usage is charged.

Format

PMP$GET_ACCOUNT_PROJECT (account, project, status)

Parameters

account: VAR of avt$account_name;

Current account name.

project: VAR of avt$project_name;

Current project name.

status: VAR of ost$status;

Status record.

Condition Identifier

None.

Remarks

The account and project names for the task are those applicable to the job in which the task executes. If the task is part of a batch job, the user can optionally specify account and project names on the LOGIN command for the job. If the task is part of an interactive job, the user cannot specify an account name or project name; the system uses the default account name and project name for the user.

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_JOB_NAMES

Purpose Returns the names of the job to which the task belongs. It returns both the job name supplied by the user and the job name supplied by the system.

Format **PMP$GET_JOB_NAMES (user_name, name, status)**

Parameters **user_name**: VAR of jmt$queue_reference_name;

User-supplied job name (1 through 31 characters).

name: VAR of jmt$job_sequence_number;

System-supplied job name (5 characters).

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks For a batch job, the user-supplied job name is the job name specified on either the LOGIN command or the SUBMIT_JOB command for the job.

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_CP_TIME

Purpose Returns the amount of central processor time the requesting task has used since it began executing. It returns both the time spent in job mode and the time spent in monitor mode.

Format **PMP$GET_TASK_CP_TIME (cp_time, status)**

Parameters **cp_time**: VAR of pmt$task_cp_time;

Number of microseconds used.

Field	Content
task_time	Time spent in job mode (0 through 7FFFFFFFFFFF hexadecimal).
monitor_time	Time spent in monitor mode (0 through 7FFFFFFFFFFF hexadecimal).

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks Successive calls to the procedure always return increasing values.

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.

PMP$GET_USER_IDENTIFICATION

Purpose Returns the user and family names of the user for whom the task executes.

Format **PMP$GET_USER_IDENTIFICATION (identification, status)**

Parameters **identification**: VAR of ost$user_identification;

User identification record.

Field	Content
user	User name (ost$user_name).
family	Family name (ost$family_name).

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks For a batch job, the user and family names are those specified on the LOGIN command for the job.

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.

PMP$MANAGE_SENSE_SWITCHES

Purpose Sets, clears, and returns sense switch settings.

Format **PMP$MANAGE_SENSE_SWITCHES (on, off, current, status)**

Parameters

on: pmt$sense_switches;

Switches to be set; set of integers, 1 through 8.

off: pmt$sense_switches;

Switches to be cleared (set of integers, 1 through 8).

current: VAR of pmt$sense_switches;

State of sense switches at procedure completion (set of integers, 1 through 8).

status: VAR of ost$status;

Status record.

Condition Identifier None.

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;
```

Job Log Messages

Each job has a job log associated with it. The purpose of the job log is to record each event during job processing so as to serve as a history and audit trail for the job.

Using the PMP$LOG procedure, a task can send a message to the job log of the job to which it belongs. A user can display the job log with the DISPLAY_LOG command (as described in the SCL System Interface manual).

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).

Operator Messages

A task can send messages to and receive messages from the system operator.

A task could send a message to an operator to inform the operator of task processing requirements or to prompt the operator for input. A message received from the operator could provide current input or could be an acknowledgment that the operator saw the message sent by the task.

Posting a job status message can keep the operator and an interactive user informed of job progress. The user or operator sees the job status message when he or she enters a DISPLAY_JOB_STATUS command.

The following procedures enable a task to communicate with the operator.

OFP$SEND_TO_OPERATOR
Sends a message to an operator.

OFP$RECEIVE_FROM_OPERATOR
Receives an operator message.

OFP$DISPLAY_STATUS_MESSAGE
Posts a job status message.

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$SEND_TO_OPERATOR

Purpose
Sends a message to the system operator.

Format
OFP$SEND_TO_OPERATOR (text, operator_id, status)

Parameters
text: string(*);
Message text.

operator_id: 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.

Condition Identifiers
ofe$message_too_long
ofe$invalid_operator_id
ofe$previous_message_not_cleared
ofe$system_unattended

Remarks

- The system assigns an action identifier to the message when it displays it at the operator terminal. The operator specifies the action identifier when responding to the message.
- Another operator message cannot be sent by the task until the operator clears the previous message. An attempt to send a message before clearing returns the OFE$PREVIOUS_MESSAGE_NOT_CLEARED condition.
- If the message is longer than OFC$MAX_SEND_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.

Condition Identifiers
ofe$message_not_available
ofe$system_unattended

Remarks

- If no operator message to the task is waiting, the call can specify that the task be suspended until it receives an operator message or that it continue executing.
- The task should prompt the operator with a message sent by an OFP$SEND_TO_OPERATOR call.
- If the operator is not logged in, the condition OFE$SYSTEM_UNATTENDED is returned.

Program Execution 3

Program Description 3-1
Program Description Structure 3-6
PMP$GET_PROGRAM_SIZE 3-7
PMP$GET_PROGRAM_DESCRIPTION 3-8
Task Parameters 3-9
Task Initiation 3-10
PMP$EXECUTE 3-11
PMP$LOAD 3-13
Task Dependencies 3-14
PMP$AWAIT_TASK_TERMINATION 3-15
PMP$TERMINATE 3-16
Task Termination 3-17
PMP$ABORT 3-18
PMP$EXIT 3-19

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)

Field	Content
contents	Set of constant identifiers indicating the information included in subsequent fields of the program description (type PMT$PROG_DESCRIPTION_CONTENTS). PMC$STARTING_PROC_SPECIFIED Starting procedure name. PMC$OBJECT_FILE_LIST_SPECIFIED Object file list. PMC$MODULE_LIST_SPECIFIED Module list. PMC$LIBRARY_LIST_SPECIFIED Object library list. PMC$LOAD_MAP_FILE_SPECIFIED Load map file name. PMC$LOAD_MAP_OPTIONS_SPECIFIED Load map options. PMC$TERM_ERROR_LEVEL_SPECIFIED Termination error level. PMC$PRESET_SPECIFIED Memory preset value. PMC$MAX_STACK_SIZE_SPECIFIED Maximum stack frame size. PMC$DEBUG_INPUT_SPECIFIED Name of the input file for a debugging program. PMC$DEBUG_OUTPUT_SPECIFIED Name of the output file for a debugging program. PMC$ABORT_FILE_SPECIFIED Abort file name. PMC$DEBUG_MODE_SPECIFIED Debug mode indicator.
starting_procedure	Name of the starting procedure (PMT$PROGRAM_NAME).

(Continued)

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)

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

Field	Content
preset_value	Value to which memory is initialized (type PMT$INITIALIZATION_VALUE). PMC$INITIALIZE_TO_ZERO Initialize all bits to zero. PMC$INITIALIZE_TO_ALL_ONES Initialize all bits to one. PMC$INITIALIZE_TO_INDEFINITE Initialize each word to floating point indefinite. PMC$INITIALIZE_TO_INFINITY Initialize each word to floating point infinite.
maximum_ stack_size	Maximum stack size (type OST$SEGMENT_LENGTH, 0 through OSC$MAX_SEGMENT_LENGTH). The system rounds the specified value up to the next page size value. The system sets a default stack size limit of two million bytes. You should increase the stack size limit only when your program fails due to a stack overflow error and the error was not caused by an infinite loop in the program.
debug_input	Name of the file containing input to a debugging program (type AMT$LOCAL_FILE_NAME).
debug_output	Name of the file to which a debugging program writes its output (type AMT$LOCAL_FILE_NAME).
abort_file	Name of the abort file (type AMT$LOCAL_FILE_ NAME).
debug_mode	Indicates whether the debugging program should control the task (type PMT$DEBUG_MODE). PMC$DEBUG_MODE_ON TRUE; Debug mode is set. Commands are read from the debug_input file and output written to the debug_output file. PMC$DEBUG_MODE_OFF FALSE; Debug mode is cleared.

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.

Program Description Structure

The program description variable is an adaptable sequence containing up to four variables. The first variable, program_attributes, determines the existence of the other variables and their size (see table 3-1).

The other variables are object_file_list, module_list, and object_library_list. If specified, each variable is an adaptable array. Each record of the object_file_list and object_library_list arrays contains a file name (type AMT$LOCAL_FILE_NAME). Each record of the module_list array contains a program name (type PMT$PROGRAM_NAME).

When initializing a program description, it is often useful to begin with the program description of the current task. The current task's program description will include a library list and load map options, and this information will most likely be of use to any new tasks initiated by the current task. The new task will then change only the information that differentiates the new task from the initiating task.

To get its own program description, a task first calls the PMP$GET_PROGRAM_SIZE procedure. The procedure returns the size of the program description, and the task then allocates a variable of that size. It then specifies the variable on a PMP$GET_PROGRAM_DESCRIPTION call. PMP$GET_PROGRAM_DESCRIPTION returns the program description of the task.

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.

PMP$GET_PROGRAM_DESCRIPTION

Purpose
Returns the program description of the requesting task.

Format
PMP$GET_PROGRAM_DESCRIPTION (program_ description, status)

Parameters
program_description: VAR of pmt$program_description;
Program description (see table 3-1).

status: VAR of ost$status;
Status record.

Condition Identifier
None.

Remarks
By getting its own program description, the task can initiate other tasks with a similar program description. The program description is specified on the PMP$EXECUTE call that initiates a task.

See the Queue Communications Example in chapter 4 for an illustration of how to use this procedure.

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.

Task Initiation

NOS/VE performs the following steps when initiating a task.

1. Loads all modules in each file in the object file list.

2. Loads each module in the module list as it is found in the program library list.

3. Loads the module containing the starting procedure, if it has not been previously loaded.

4. Loads modules from the program library to satisfy external references in the loaded modules.

5. Checks whether the PMC$DEBUG_MODE_ON Debug mode indicator is specified. If it is, the system calls the debugging program to set up a debugging environment for the task.

6. Initializes the status record specified as the task_status parameter on the PMP$EXECUTE call. The status record is initialized for normal completion.

7. Begins program execution by calling the starting procedure if one is specified in the program description. Otherwise, if no starting procedure is specified, it calls the last transfer symbol encountered during loading. The newly initiated task run asynchronously from the task which invoked it when OSC$NOWAIT is specified as the wait parameter on the PMP$EXECUTE call.

Additional modules can be loaded from the program library list during program execution. To load the module, the task first calls the PMP$LOAD procedure to determine the module address. This dynamic loading mechanism is useful in the following instances.

- When the program name for a module is unknown before the task begins.

- When a conditional load is appropriate because the module is not always needed.

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).

task_id: VAR of pmt$task_id;

Task identifier supplied by the system (0 through PMC$MAX_TASK_ID).

task_status: VAR of pmt$task_status;

Status record of initiated task.

Field	Content
complete	Indicates whether the initiated task has completed (boolean). TRUE Task complete. FALSE Task incomplete.
status	Status record returned by the completed task (type OST$STATUS).

NOTE

If OSC$NOWAIT is specified as the wait parameter on this call, you must declare the task_status variable as a STATIC variable; if not, the system would discard it if the procedure containing it terminated.

status: VAR of ost$status;

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

Condition Identifiers

pme$invalid_file_name
pme$invalid_list_length
pme$invalid_preset_option
pme$invalid_stack_size_option
pme$invalid_term_error_level
pme$invalid_wait_parameter
pme$map_option_conflict
pme$prog_description_too_small

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.

Task Dependencies

As described in the PMP$EXECUTE procedure description, the initiating task can specify whether it is suspended while the initiated task executes. If the initiating task suspends itself, the task does not resume until the initiating task and all its initiated tasks complete.

For example, suppose TASKA initiates TASKB and suspends itself. TASKB then initiates TASKC, and TASKB then suspends itself. TASKC must terminate before TASKB can resume execution and, in the same manner, TASK B must terminate before TASKA can resume.

A task could also specify that its processing continue after it initiates a task. Later, in its processing, it could call the procedure PMP$AWAIT_TASK_TERMINATION to suspend itself until the specified task, and any other tasks initiated by the specified task, have terminated.

For example, TASKA can continue processing after initiating TASKB. TASKA then calls PMP$AWAIT_TASK_TERMINATION to suspend itself until TASKB terminates. If TASKB has initiated TASKC, TASKA must then also wait for termination of TASKC.

The task specified on the PMP$AWAIT_TASK_TERMINATION call must have been initiated by the calling task. For this purpose, the task identifier specified on the call is returned by the PMP$EXECUTE call to the calling task.

A task that specifies that its processing continue after it initiates a task can terminate the task it initiated. It does so by specifying the task identifier on a PMP$TERMINATE procedure call.

PMP$TERMINATE terminates the specified task and any tasks initiated by the terminated task. For example, if TASKA initiates TASKB and TASKB initiates TASKC, a PMP$TERMINATE call by TASKA to terminate TASKB also terminates TASKC.

The status record returned to the terminating task is the status record initialized when it initiated the task. If the task has changed the contents of the status record, the information is passed back to the initiating task. For example, if TASKA initiates TASKB and TASKB sets its status record to indicate abnormal status, the abnormal status record is returned to TASKA when TASKA terminates TASKB.

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

PMP$TERMINATE

Purpose Terminates an initiated task.

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

Parameters **task_id**: pmt$task_id;

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

status: VAR of ost$status;

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

Condition Identifiers
pme$invalid_task_id
pme$task_not_current_child

Remarks The terminated task must be a task that was initiated by the requesting task. Any tasks initiated by the terminated task are also terminated.

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$ABORT

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

Format **PMP$ABORT (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 PMP$ABORT calls the debugging program to execute the contents of the abort file, if an abort file was specified in the program description.

This procedure is used to terminate the task when it cannot perform its function due to its own error.

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.

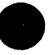
Task Communication 4

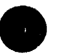
General Wait . . . 4-1
OSP$AWAIT_ACTIVITY_COMPLETION . . . 4-2

Job-Local Queues . . . 4-4
PMP$DEFINE_QUEUE . . . 4-5
PMP$REMOVE_QUEUE . . . 4-6
PMP$CONNECT_QUEUE . . . 4-7
PMP$DISCONNECT_QUEUE . . . 4-8
PMP$RECEIVE_FROM_QUEUE . . . 4-9
PMP$SEND_TO_QUEUE . . . 4-11

Queue Information Retrieval . . . 4-12
PMP$GET_QUEUE_LIMITS . . . 4-13
PMP$STATUS_QUEUE . . . 4-14
PMP$STATUS_QUEUES_DEFINED . . . 4-15

Queue Communication Example . . . 4-16





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.

OSP$AWAIT_ACTIVITY_COMPLETION

Purpose Suspends the requesting task until completion of one of the activities specified in the wait list.

Format **OSP$AWAIT_ACTIVITY_COMPLETION (wait_list, ready_index, status)**

Parameters **wait_list**: ost$wait_list;

List of one or more events that must occur before the task resumes (adaptable array of OST$ACTIVITY records). The list must specify at least one event. Each record has the following fields.

Field	Content
activity	Key field indicating the activity type (type OST$WAIT_ACTIVITY). OSC$AWAIT_TIME Time period specified in the milliseconds field. PMC$AWAIT_TASK_TERMINATION Task specified in the task_id field. PMC$AWAIT_LOCAL_QUEUE_MESSAGE Queue connection specified in the qid field.
milliseconds	Number of milliseconds that must elapse (0 through 0FFFFFFFF hexadecimal).
task_id	Identifier of the task that must complete (type PMT$TASK_ID, as returned by the PMP$EXECUTE call).
qid	Identifier of the queue from which a message must be received (type PMT$QUEUE_CONNECTION as returned by the PMP$CONNECT_QUEUE call).

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

Job-Local Queues

Tasks within a job can send messages to each other via a queue that is local to the job. A job-local queue is created when a task within the job defines the queue with a PMP$DEFINE_QUEUE call. The PMP$DEFINE_QUEUE call names the queue and specifies the usage bracket and removal bracket for the queue. The usage bracket is the range of rings in which a task can connect to the queue. The removal bracket is the range of rings in which a task can remove the queue definition.

After a queue is defined, any task within the job executing within the usage bracket of the queue can connect to the queue. To connect to the queue, the task specifies the queue name on a PMP$CONNECT_QUEUE call. The PMP$CONNECT_QUEUE procedure returns a connection identifier to the task. The task can then send and receive queue messages using the connection identifier on PMP$SEND_TO_QUEUE and PMP$RECEIVE_FROM_QUEUE calls.

When a task sends a message, it cannot specify the task to receive the message. A message sent to a queue is received by the next task connected to the queue that executes a PMP$RECEIVE_FROM_QUEUE call.

The connection of a task to a queue ends when the task terminates or when it executes a PMP$DISCONNECT_QUEUE call that specifies the connection identifier. A queue is discarded when the job in which it was defined terminates or when a task within the job (executing within the removal bracket for the queue) removes the queue definition. To remove a queue definition, the task specifies the queue name on a PMP$REMOVE_QUEUE call.

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$REMOVE_QUEUE

Purpose Removes a queue definition.

Format **PMP$REMOVE_QUEUE (name, status)**

Parameters **name**: pmt$queue_name;

Queue name as defined by a PMP$DEFINE_QUEUE call.

status: VAR of ost$status;

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

Condition Identifiers

pme$incorrect_queue_name
pme$nonempty_queue
pme$removal_bracket_error
pme$task_connected_to_queue
pme$unknown_queue_name

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$DISCONNECT_QUEUE

Purpose Disconnects the task from a queue.

Format **PMP$DISCONNECT_QUEUE (qid, status)**

Parameters **qid**: pmt$queue_connection;

Queue connection identifier returned by the PMP$CONNECT_QUEUE call.

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$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.

message: VAR of pmt$message;

Message received from 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$error_segment_privilege
pme$unknown_queue_identifier
pme$usage_bracket_error

Remarks

The received message is removed from the queue.

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

Queue Information Retrieval

A task can get the following information about the defined queues of its job.

PMP$STATUS_QUEUE

Number of tasks connected to the queue, number of messages waiting to be received from the queue, and number of tasks waiting to receive a message from the queue.

PMP$STATUS_QUEUES_DEFINED

Number of queues currently defined for the job.

PMP$GET_QUEUE_LIMITS

Maximum number of queues in a job, maximum number of tasks connected to a queue, and maximum number of messages waiting in a queue.

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_QUEUE

Purpose Returns the number of tasks connected to the queue, the number of queued messages, and the number of waiting tasks.

Format **PMP$STATUS_QUEUE (qid, counts, status)**

Parameters **qid**: pmt$queue_connection;

Queue connection identifier returned by the PMP$CONNECT_QUEUE call.

counts: VAR of pmt$queue_status;

Queue status record having the following fields:

connections

Number of tasks connected to the queue (type PMT$CONNECTED_TASKS_PER_QUEUE, 0 through PMC$MAX_QUEUES_PER_JOB).

messages

Number of queue messages (type PMT$MESSAGES_PER_QUEUE, 0 through PMC$MAX_MESSAGES_PER_QUEUE).

waiting_tasks

Number of waiting tasks (type PMT$CONNECTED_TASKS_PER_QUEUE, 0 through PMC$MAX_QUEUES_PER_JOB).

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.

Queue Communication Example

The following example illustrates use of the queue communication calls described in this chapter and program execution calls described in chapter 3.

The example comprises the source text for two modules. The first module is for the control task; the second is for the worker task. The control task starts execution of the worker task and communicates with it via a queue and a shared segment access file.

```
MODULE try_queues_control_task;

??PUSH (LISTEXT := ON)??
*copyc PMP$DEFINE_QUEUE
*copyc PMP$CONNECT_QUEUE
*copyc PMP$SEND_TO_QUEUE
*copyc PMP$RECEIVE_FROM_QUEUE
*copyc AMP$OPEN
*copyc AMP$GET_SEGMENT_POINTER
*copyc PMP$EXECUTE
*copyc PMP$GET_PROGRAM_SIZE
*copyc PMP$GET_PROGRAM_DESCRIPTION
??POP??

{ This program demonstrates how two tasks can }
{ communicate using a shared segment to hold the }
{ data and a local queue to pass the relative }
{ address of an item in the shared segment. }

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

```
{ 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 program description for the worker task }
{ using the program description of the control task }
{ as a base. }

PMP$GET_PROGRAM_SIZE (number_of_object_files,
  number_of_modules, number_of_libraries, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Allocates a sequence long enough for the program }
{ attributes variable, the object file list, the }
{ module list, and the object library list. }

PUSH worker_program:
  [[REP (#SIZE (pmt$program_attributes) +
  (number_of_object_files * #SIZE(amt$local_file_name))
  + (number_of_modules * #SIZE (pmt$program_name)) +
  (number_of_libraries * #SIZE (amt$local_file_name)))
  OF cell]];

{ Save returns the program description of the control task. }

PMP$GET_PROGRAM_DESCRIPTION (worker_program^, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Adds the starting procedure to the program }
{ description.  The name of the starting procedure is }
{ WORKER_TASK. }

RESET worker_program;
NEXT worker_program_attributes IN worker_program;
worker_program_attributes^.contents :=
  worker_program_attributes^.contents +
  $pmt$prog_description_contents
  [pmc$starting_proc_specified];
worker_program_attributes^.starting_procedure :=
  'WORKER_TASK';
```

```
{ 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;
```

```
{ Start the worker task. The osc$nowait parameter }
{ indicates that the worker task and the control task }
{ are to execute at the same time. }

PMP$EXECUTE (worker_program^,
  worker_program_parameters^,
  osc$nowait, task_id,
  task_status, status);
IF NOT status.normal THEN
    RETURN;
IFEND;
PROCEND execute_worker_task;

VAR
{ The following variables define segment }
{ characteristics.}

{ File attributes: read, shorten, append, and modify }
{ access modes and undefined (U) record type. }
shared_segment_attributes:
  [STATIC] array [1 ..2] of amt$file_item :=
  [[amc$access_mode, $pft$usage_selections [pfc$read,
  pfc$shorten, pfc$append, pfc$modify]],
  [amc$record_type, amc$undefined]],

{ File name: SHARED_COMMUNICATION_SEGMENT }
shared_segment_name: [STATIC] amt$local_file_name :=
  'shared_communication_segment',

shared_segment_id: amt$file_identifier,
shared_segment_pointer: amt$segment_pointer,
shared_heap: ^HEAP ( * ),

{ The following variables define the job local queue. }

communication_queue_name: [STATIC] pmt$queue_name :=
  'communication_queue',
communication_queue: pmt$queue_connection,

{ Values for the following variables are returned }
{ when the worker task is started. }

worker_task_status: pmt$task_status,
worker_task_id: pmt$task_id,
```

```
{ 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;
```

```
{ Calls the procedure to start the worker task. }

execute_worker_task (shared_segment_name,
  shared_segment_attributes, communication_queue_name,
  worker_task_id, worker_task_status, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

{ Allocates an eight-character data item in the }
{ shared segment. }

RESET shared_heap^;
message_to_worker.contents := pmc$message_value;
ALLOCATE worker_text_pointer IN shared_heap^;
worker_text_pointer^ := 'Hello!';

{ Builds a relative pointer to the data item and }
{ puts it in the message to be sent to the worker }
{ task. }

message_to_worker_value_pointer :=
  ^message_to_worker.value;
RESET message_to_worker_value_pointer;
NEXT worker_text_relative_ptr_ptr IN
  message_to_worker_value_pointer;
worker_text_relative_ptr_ptr^ :=
  #REL (worker_text_pointer, shared_heap^);

{ Sends the message. }

PMP$SEND_TO_QUEUE (communication_queue,
  message_to_worker, status);
IF NOT status.normal THEN
  RETURN;
IFEND;

PROCEND control_task;
MODEND try_queues_control_task;
```

```
{ 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;
```

```
{ Open and get a pointer to the segment used to pass }
{ information between the control task and the }
{ worker task. }

NEXT shared_segment_name IN worker_parameters;
NEXT number_of_shared_attributes IN worker_parameters;
NEXT shared_segment_attributes:
  [1 .. number_of_segment_attributes^] IN
  worker_parameters;
AMP$OPEN (shared_segment_name^, amc$segment,
  shared_segment_attributes, shared_segment_id,status);
IF NOT status.normal THEN
  RETURN;
IFEND;
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;

{ Connect to communication queue. }

NEXT communication_queue_name IN worker_parameters;
PMP$CONNECT_QUEUE (communication_queue_name^,
  communication_queue, status);
IF NOT status.normal THEN
  RETURN;
IFEND;
```

```
{ 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

System Condition Detection 5-1

PMP$ENABLE_SYSTEM_CONDITIONS 5-2

PMP$INHIBIT_SYSTEM_CONDITIONS 5-3

Condition Handling 5-4

Condition Handler Establishment 5-6

PMP$ESTABLISH_CONDITION_HANDLER 5-9

PMP$DISESTABLISH_COND_HANDLER 5-11

Condition Handler Processing 5-12

System Condition Handler 5-13

PMP$CONTINUE_TO_CAUSE 5-15

Block Exit Processing Condition Handler 5-16

Interactive Condition Handler 5-17

Job Resource Condition Handler 5-17

Segment Access Condition Handler 5-18

Process Interval Timer Condition Handler 5-19

PMP$SET_PROCESS_INTERVAL_TIMER 5-20

User-Defined Condition Handler 5-21

PMP$CAUSE_CONDITION 5-22

PMP$TEST_CONDITION_HANDLER 5-23

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$ENABLE_SYSTEM_CONDITIONS

Purpose Enables detection of a specified system condition.

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

Parameters **conditions**: pmt$system_conditions;

Condition set enabled. The set can contain any of the following constant 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

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

Condition Handling

When a condition occurs, NOS/VE executes the condition handler in effect for that condition. A condition handler is a procedure that determines further processing after a condition occurs. If no condition handler is in effect for the condition, NOS/VE executes its standard procedure for the condition category. Table 5-2 lists the standard condition processing for each category.

A condition handler is in effect for a condition if it meets the following qualifications.

- It is the most recently established condition handler for the condition.
- It has not been disestablished by a PMP$DISESTABLISH_COND_ HANDLER call.
- The condition is within the scope of the condition handler. The scope of a condition handler depends on the category of the conditions for which it is established. Table 5-3 lists the scope of a condition handler for each condition category.

A call to disestablish a condition handler must be within the scope of the condition handler.

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.

Condition Handler Establishment

To establish a condition handler for a condition or set of conditions, a task calls the PMP$ESTABLISH_CONDITION_HANDLER procedure. The procedure call must be in the outermost block of the blocks in which the condition handler is to be effective.

The condition set is specified by the conditions parameter on the call. The selector field of the parameter specifies the type of condition set. The possible condition sets follow.

PMC$ALL_CONDITIONS

All conditions except unselectable system conditions, interactive conditions, and the process interval timer condition.

PMC$CONDITION_COMBINATION

All conditions within one or more categories.

A category identifier

One or more conditions within a category.

Table 5-4 lists the fields generated for each selector value and the content of those fields.

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.

Table 5-5. Selectable System Conditions

Condition Identifier	Meaning
PMC$ADDRESS_SPECIFICATION	Address specification error.
PMC$ACCESS_VIOLATION	Memory access failed due to lack of access permission.
PMC$ARITHMETIC_OVERFLOW	Arithmetic overflow in result.
PMC$ARITHMETIC_SIGNIFICANCE	Arithmetic significance loss in result.
PMC$DETECTED_UNCORRECTED_ERR	Detected uncorrectable error.
PMC$DIVIDE_FAULT	Divide fault.
PMC$ENVIRONMENT_SPECIFICATION	Environment specification error.
PMC$EXPONENT_OVERFLOW	Exponent overflow in result.
PMC$EXPONENT_UNDERFLOW	Exponent underflow in result.
PMC$FP_INDEFINITE	Floating point indefinite result.
PMC$FP_SIGNIFICANCE_LOSS	Floating point significance loss in result.
PMC$INSTRUCTION_SPECIFICATION	Instruction specification error.
PMC$INTER_RING_POP	Attempt to pop stack frames across a ring boundary.
PMC$INVALID_BDP_DATA	Invalid BDP data.
PMC$INVALID_SEGMENT_RING_0	Either invalid segment number or ring number set to zero.
PMC$OUT_CALL_IN_RETURN	Processor attempted to call a less privileged call from a more privileged procedure or return to a more privileged procedure from a less privileged procedure.
PMC$PRIVILEGED_INSTRUCTION	Improper attempt to execute a privileged instruction.
PMC$UNIMPLEMENTED_INSTRUCTION	Instruction code unimplemented for this processor.

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

Remarks

- The specified procedure remains in effect for the condition set until the task exits the establishing block or the procedure is explicitly disestablished.
- The establish_descriptor parameter on the call must specify a pointer to a variable allocated within the current stack frame. The system uses the space to maintain internal condition handler information.

  NOTE

  The variable to which the establish_descriptor parameter points must be unique for each PMP$ESTABLISH_CONDITION_HANDLER call in the procedure.

- The task can use either of the following methods to allocate the descriptor variable and pointer.
  - The task can declare the descriptor variable (descriptor: pmt$established_handler) and then specify a pointer to the variable in the procedure call (descriptor^).
  - The task can declare a pointer (descrip_ptr: ^pmt$established_handler), use a PUSH statement to allocate the space needed (PUSH descrip_ptr;), and specify the pointer in the procedure call (descrip_ptr).

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

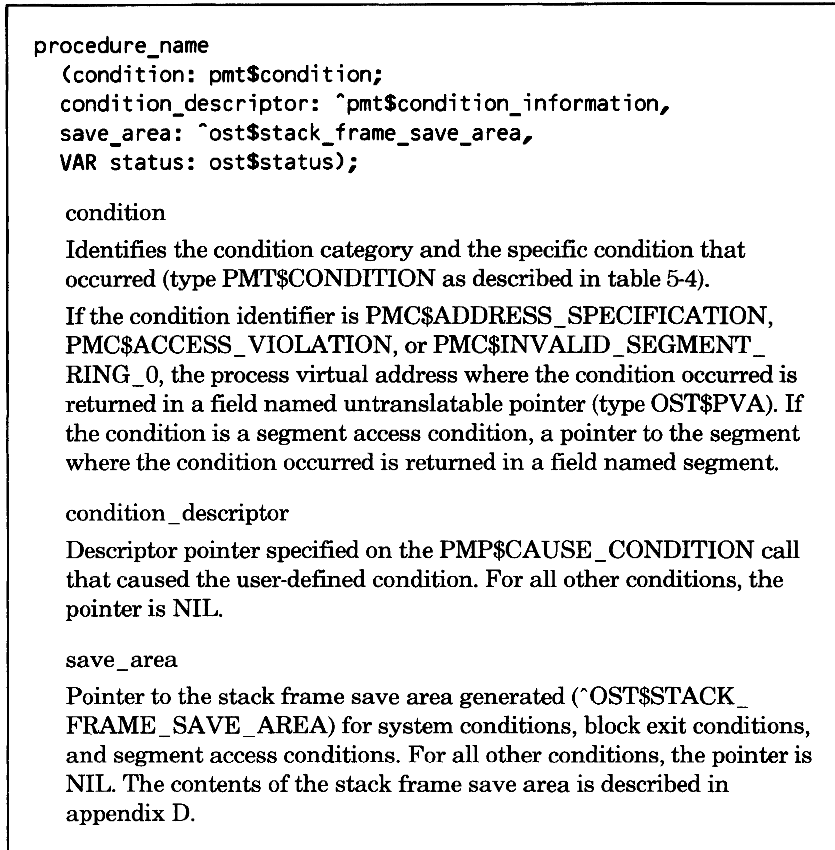
Remarks

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.

Condition Handler Processing

A condition handler procedure declaration must have the format shown in figure 5-1. When NOS/VE calls a condition handler, it passes the information described in the parameter list.

At its termination, the condition handler returns its completion status in the status record. If the condition handler returns abnormal status, the task terminates. If it returns normal status, the task resumes.



```
procedure_name
  (condition: pmt$condition;
  condition_descriptor: ^pmt$condition_information,
  save_area: ^ost$stack_frame_save_area,
  VAR status: ost$status);
```

condition

Identifies the condition category and the specific condition that occurred (type PMT$CONDITION as described in table 5-4).

If the condition identifier is PMC$ADDRESS_SPECIFICATION, PMC$ACCESS_VIOLATION, or PMC$INVALID_SEGMENT_RING_0, the process virtual address where the condition occurred is returned in a field named untranslatable pointer (type OST$PVA). If the condition is a segment access condition, a pointer to the segment where the condition occurred is returned in a field named segment.

condition_descriptor

Descriptor pointer specified on the PMP$CAUSE_CONDITION call that caused the user-defined condition. For all other conditions, the pointer is NIL.

save_area

Pointer to the stack frame save area generated (^OST$STACK_FRAME_SAVE_AREA) for system conditions, block exit conditions, and segment access conditions. For all other conditions, the pointer is NIL. The contents of the stack frame save area is described in appendix D.

Figure 5-1. Condition Handler Procedure Declaration

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

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

- Change the save area to correct the cause of the condition and to return normal status.
- Change the save area to circumvent the offending instruction and return normal status.
- Perform an explicit pop or nonlocal exit to circumvent the offending environment.
- Terminate the task by calling the PMP$EXIT or PMP$ABORT procedure or by returning abnormal status.
- Pass the condition to the next most recently established condition handler for the condition by calling the PMP$CONTINUE_TO_CAUSE procedure. If the condition is not within the scope of the next most recently established condition handler, NOS/VE processes the condition as if no condition handler is in effect for the condition.

If the condition handler returns normal status, execution resumes using the contents of the saved stack frame area. If the stack frame area was not modified to correct the condition or to circumvent the offending instruction, the condition immediately recurs and NOS/VE calls the condition handler again.

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.

Block Exit Processing Condition Handler

A block exit occurs when execution of the current procedure terminates. Any of the following events terminate the current procedure.

- Procedure execution completes.
- A nonlocal EXIT statement terminates the block containing the block exit condition handler.
- A PMP$EXIT call is executed.
- A PMP$ABORT call is executed.

Each event causes the popping of the task stack. Procedure completion removes only the topmost stack frame. A nonlocal exit could pop several stack frames. PMP$EXIT and PMP$ABORT remove each stack frame in succession.

To ensure that the establishing block cannot be circumvented without notification, a task can establish a block exit processing condition handler. The condition handler can perform cleanup operations for its block.

Before removing a stack frame, the system hardware checks whether a block exit processing condition handler is in effect for the block. If one is in effect, the system executes the condition handler before removing the stack frame.

NOTE

Block exit notification cannot be ensured if the stack environment is not intact.

The system passes a block exit processing condition handler a pointer to the stack frame save area for its establishing block. The P register of the stack frame save area always points to the return or pop instruction that caused the condition.

When a block exit processing condition handler returns normal status, the task resumes; the condition will not recur regardless of the condition handler processing.

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.

Segment Access Condition Handler

A segment access condition indicates an abnormality in the mass storage backing up a segment.

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

If a segment access condition occurs while a condition handler is in effect, NOS/VE passes the condition handler a pointer to the stack frame save area of the block where the segment access condition occurred. The P register in the stack frame save area points to the instruction that caused the condition.

A segment access condition handler must process the condition so that it does not recur. If it returns normal status without taking appropriate action, the condition immediately recurs and NOS/VE calls the condition handler again.

The following are possible ways that a condition handler could process a segment access condition.

- Correct the condition cause and return normal status.
- Perform an explicit pop or nonlocal exit to circumvent the offending environment.
- Terminate the task with a PMP$EXIT or PMP$ABORT call or by returning abnormal status.

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.

PMP$SET_PROCESS_INTERVAL_TIMER

Purpose Sets the process interval timer.

Format **PMP$SET_PROCESS_INTERVAL_TIMER (microseconds, status)**

Parameters **microseconds**: pmt$pit_value;

Value placed in timer (PMC$MINIMUM_PIT_VALUE through PMC$MAXIMUM_PIT_VALUE).

status: VAR of ost$status;

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

Condition Identifier pme$pit_value_out_of_range

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$CAUSE_CONDITION

Purpose

Causes a user-defined condition. It interrupts the task and executes the condition handler in effect for the specified condition if one exists. If no condition handler is in effect for the condition, the PMP$CAUSE_CONDITION call has no effect and the task resumes.

Format

PMP$CAUSE_CONDITION (condition_name, condition_descriptor, status)

Parameters

condition_name: pmt$condition_name;

Condition name.

condition_descriptor: ^pmt$condition_information;

Pointer passed to the condition handler.

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$incorrect_condition_name
pme$invalid_condition_handler
pme$no_established_handler
pme$stack_overwritten

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

Status Record Generation 6-1

Status Parameters 6-2
OSP$SET_STATUS_ABNORMAL 6-3
OSP$APPEND_STATUS_PARAMETER 6-4
OSP$APPEND_STATUS_INTEGER 6-5
OSP$SET_STATUS_FROM_CONDITION 6-6

Status Severity Check 6-7

OSP$GET_STATUS_SEVERITY 6-9

Message Formatting 6-10

Message Levels 6-11
OSP$FORMAT_MESSAGE 6-12
OSP$GET_MESSAGE_LEVEL 6-14
OSP$SET_MESSAGE_LEVEL 6-15

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.

Status Parameters

The text string specified in a status record consists of status parameters. A status parameter is an item of information inserted into the message template to make the message meaningful. For example, it could be the name of the file for which the error is detected or it could be a string whose syntax is incorrect.

The first character of the string in the text field is the delimiter character, OSC$STATUS_PARAMETER_DELIMITER. The delimiter character separates status parameters within the string.

The OSP$SET_STATUS_ABNORMAL procedure performs general status record initialization. The text string specified on the call is processed as the first status parameter for the record. Subsequent status parameters can be appended to the record with OSP$APPEND_STATUS_PARAMETER and OSP$APPEND_STATUS_INTEGER calls.

The AMP$SET_FILE_INSTANCE_ABNORMAL performs status record initialization according to file interface conventions. The text string specified on the call is processed as the eighth status parameter. The procedures initialize the first seven status parameters with the following information:

1. Local file name.
2. Name of the file interface request that detected the condition.
3. Access level (record or segment access).
4. File organization.
5. Record type.
6. Block type.
7. Reserved for internal system use.

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_PARAMETER

Purpose Appends a status parameter to the text of a status record.

Format **OSP$APPEND_STATUS_PARAMETER (delimiter, text, status)**

Parameters **delimiter**: char;

First character appended. If the character matches the first character of the text field (usually 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.

text: string (*);

Status parameter text.

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

Status record to which the text is appended.

Condition Identifier None.

Remarks

- OSP$APPEND_STATUS_PARAMETER 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_PARAMETER 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.

OSP$SET_STATUS_FROM_CONDITION

Purpose

Initializes an abnormal status record within a condition handler procedure using the information passed to the condition handler.

Format

OSP$SET_STATUS_FROM_CONDITION (identifier, condition, save_area, condition_status, status)

Parameters

identifier: string (2);

Two-character process identifier.

condition: pmt$condition;

Condition selector record passed to the condition handler identifying the condition that has occurred. The fields of the record are described in table 5-4.

save_area: ^ost$stack_frame_save_area;

Pointer to the stack frame area.

condition_status: VAR of ost$status;

Initialized status record.

status: VAR of ost$status;

Status record. The operating system process identifier, OS, is returned.

Condition Identifiers

ose$empty_block_exit_reason
ose$empty_system_condition
ose$invalid_condition_selector
ose$invalid_save_area
ose$unknown_interactive_cond
ose$unknown_segment_condition

Remarks

- A condition handler procedure must return a status record. The condition handler is called when a condition for which it is established is detected. Information describing the condition is passed to the condition handler.

- OSP$SET_STATUS_FROM_CONDITION can construct a status record only for a specific condition. It cannot construct a status record when the condition selector that is passed to it is either PMC$ALL_CONDITIONS or PMC$CONDITION_COMBINATION.

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

(Continued)

```
{ If the entry name cannot be loaded, procedure PMP$LOAD }
{ returns an abnormal status.  The message associated with }
{ the abnormal status is formatted and put into the job log. }

   pmp$load (entry_name, pmc$procedure_address,
       entry_address, stat);
   IF NOT stat.normal THEN
     osp$get_status_severity (stat.condition, severity,
         ignore_status);
     IF severity > osc$warning_status THEN
       osp$format_message (stat, osc$full_message_level,
           max_line_size, message, ignore_status);
       pointer := ^message;
       RESET pointer;
       NEXT msg_line_count IN pointer;
       FOR i := 1 TO msg_line_count^ DO
         NEXT msg_line_size IN pointer;
         NEXT msg_line_text: [msg_line_size^] IN pointer;
         pmp$log (msg_line_text^, ignore_status);
       FOREND;
     IFEND;
   IFEND;

   PROCEND sample;
 MODEND sample;
```

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 Formatting

Using a status record, a task can generate a standard error message with the OSP$FORMAT_MESSAGE procedure. The procedure uses the condition code in the status record and the message level specified on the call to find the message template. The standard message templates are listed in the Diagnostic Messages manual.

If no message template exists for the condition code, OSP$FORMAT_ MESSAGE returns the contents of the status record within the following line.

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

The status message generated by OSP$FORMAT_MESSAGE could be more than one line of information. The OSP$FORMAT_MESSAGE call specifies the maximum characters in a message line. If the message exceeds the maximum characters in a line, the message is split into lines. If possible, each line ends after a delimiter character; otherwise, two dots are appended to the line indicating continuation.

OSP$FORMAT_MESSAGE returns the message as a sequence beginning with the number of lines in the message followed by the message lines. Each line begins with the line length followed by the message text.

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.

OSP$FORMAT_MESSAGE

Purpose Returns a status message.

Format **OSP$FORMAT_MESSAGE (message_status, message_level, max_message_line, message, status)**

Parameters **message_status**: ost$status;

Status record.

message_level: ost$status_message_level;

Message level.

OSC$CURRENT_MESSAGE_LEVEL

Current setting.

OSC$BRIEF_MESSAGE_LEVEL

Brief mode.

OSC$FULL_MESSAGE_LEVEL

Full mode.

OSC$EXPLAIN_MESSAGE_LEVEL

Currently, the same as full mode.

max_message_line: ost$max_status_message_line;

Maximum number of characters in a message line (OSC$MIN_MESSAGE_LINE, 32, through OSC$MAX_MESSAGE_LINE, 256).

message: VAR of ost$status_message;

Message sequence. The sequence begins with a variable of type OST$STATUS_MESSAGE_LINE_COUNT containing the number of lines in the message followed by the message lines.

status: VAR of ost$status;

Status record.

Condition Identifier None.

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$GET_MESSAGE_LEVEL

Purpose Returns the current message level of the job.

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

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

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.

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

Creating a NOS Job 7-1
- Creating the Command Variable 7-2
  - Command Variable Content 7-2
  - Command Record Size Limit 7-3
- Requesting the Link File 7-4
- Assigning Link File Attributes 7-5

Starting a NOS Job 7-6

Communication Between the Task and Job 7-7
- Link File Deadlock 7-7
- Data Conversion 7-7
- Sending Data to the NOS Job 7-8
- Receiving Data from the NOS Job 7-8
- Fetching Information About the Link File 7-9
- Positioning the Link File 7-9
- Unsupported File Interface Calls 7-9

NOS Job Communication with the NOS/VE Task 7-10
- Subroutine Calling Convention 7-10
- NOS Link Subroutines 7-11
  - OPENLNK Subroutine 7-11
  - CLOSLNK Subroutine 7-12
  - GETNLNK Subroutine 7-13
  - GETPLNK Subroutine 7-14
  - PUTNLNK Subroutine 7-15
  - PUTPLNK Subroutine 7-15
  - WREPLNK Subroutine 7-16

Interstate Communication Example 7-17

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.

Creating the Command Variable

Before starting a NOS job, you must store the NOS command record for the job in the NOS/VE command variable. The NOS/VE command variable is an SCL string variable. The following are the SCL commands and CYBIL procedure calls that can prepare the NOS command record.

Function	SCL Command	Procedure Call
Declare a string variable.	CREATE_VARIABLE command	CLP$CREATE_ VARIABLE call
Store NOS commands in the string variable.	SCL assignment statement	CLP$WRITE_ VARIABLE call

Command Variable Content

The NOS job defined within the NOS/VE command variable can have only one record, the NOS command record. Any additional input must be read from other NOS files. The NOS commands and programs must not reference file INPUT.

The NOS commands stored in the command variable need not include USER or CHARGE commands. The system adds USER and CHARGE commands to the command record using the default information for the user name or the information specified on a SET_LINK_ATTRIBUTES command (described in the SCL System Interface manual).

NOTE

To use the link subroutines within the NOS job, the NOS user name must have the CUCP and CNVE bits of its access word (bits 11 and 18, respectively) set. If the user name does not have the proper validation, NOS aborts the job when it attempts to call a link subroutine.

The first NOS command in the NOS/VE command variable must be the job command. Each command must end with a period or right parenthesis.

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.

Requesting the Link File

A NOS/VE job defines a link file with the REQUEST_LINK command.

The file on the command must not be already assigned to another device class; if it is, the command returns an error status and terminates.

The following is the SCL command format.

REQUEST_LINK
FILE=local file
STATUS=status variable

FILE

Local file name. This parameter is required.

STATUS

Optional status variable.

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

Starting a NOS Job

A NOS/VE task starts a NOS job by opening the link file. It opens the file with an AMP$OPEN call. The AMP$OPEN call can also specify the user_ info attribute value.

When opening a link file, the access_level parameter value must be AMC$RECORD.

While the link file is open, the NOS/VE task can communicate with the started NOS job. Closing the link file prevents further communication with the NOS job. The NOS job continues executing until its termination.

The following are additional exception conditions that can be returned by an AMP$OPEN call that specifies a link file.

ICE$ACCESS_LEVEL_NOT_RECORD

The access level attribute of the link file is not AMC$RECORD.

ICE$EMPTY_JOB_SPEC_VARIABLE

The specified command variable is empty.

ICE$LINK_IS_ALREADY_OPEN

A link file is already open within this job; only one instance of open of a link file is allowed at a time for a job.

ICE$NO_JOB_SPEC_VARIABLE

The user_info attribute does not specify a command variable name.

ICE$PARTNER_CANNOT_BE_STARTED

The NOS job could not be started due to a job error (such as a job command or user validation error).

ICE$PARTNER_JOB_TOO_LONG

The NOS command variable was too large to be sent to NOS. See Command Record Size Limit in this chapter.

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.

Sending Data to the NOS Job

If the access_mode attribute for the link file includes PFC$SHORTEN, PFC$APPEND, or PFC$MODIFY, the NOS/VE task can send data to the NOS job. To do so, it puts data in the link file with AMPPUT_NEXT, AMPPUT_PARTIAL, or AMP$PUT_DIRECT calls. It can also write a partition delimiter in the link file with an AMP$WRITE_END_PARTITION call. The byte_address parameter on these calls is not used.

For AMP$PUT_PARTIAL calls, a partial record is sent for each call. The record transfer is completed when the AMC$TERMINATE term_option is issued.

The calls return abnormal status (ICE$WRITE_DEADLOCK) if the NOS job is writing to the link file when the put call attempts to write to the file. The calls also return abnormal status (ICE$PARTNER_ENDED) if the NOS job has terminated or has closed the link file.

Receiving Data from the NOS Job

If the access_mode attribute for the link file includes PFC$READ, the NOS/VE task can receive data from the NOS job. To do so, it gets data from the link file with AMPGET_NEXT, AMPGET_PARTIAL, or AMP$GET_DIRECT calls.

If the file_position returned by the get call is AMC$EOP, the call reads a partition delimiter sent by the NOS job.

The byte_address parameter on the calls is not used.

A get call returns abnormal status (ICE$READ_DEADLOCK) if the NOS job is also attempting to read from the link file. A get call also returns abnormal status (ICE$PARTNER_ENDED) if the NOS job has terminated or has closed the link file.

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 Job Communication with the NOS/VE Task

The started NOS job communicates with the NOS/VE task that started it by reading and writing data to the link file. It reads and writes data using communication subroutines stored on a system library.

NOTE

The NOS job must include the libraries containing the link subroutines within its load library. To do so, it must execute the following command.

```
LIBRARY,NVELIB/A.
```

Before it can read and write to the link file, the NOS job must open the link file with an OPENLNK subroutine call. It must close the link file with a CLOSLNK call within the same job step.

Because the NOS/VE task waits for communication from the started NOS job, it is recommended that the NOS job step that communicates with the NOS/VE task execute as soon as possible within the NOS job. After the link file is closed, the NOS/VE task and the NOS job can continue their separate processing.

Subroutine Calling Convention

The NOS link subroutines use the standard NOS product set calling convention. Therefore, FORTRAN programs can call the subroutines directly (as shown in the following example).

COMPASS programs can also call the subroutines. The address of the parameter list is passed in register A1. The parameter list consists of the argument addresses ending with an address of zero.

For example, the following is a FORTRAN declaration and call for the GETNLNK subroutine.

```
INTEGER ARR(20), RECLENG, UNUSED, POSIT, STATUS
CALL GETNLNK (ARR, 20, RECLENG, UNUSED, POSIT, STATUS)
```

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.

CLOSLNK Subroutine

The CLOSLNK subroutine closes the link file, preventing further reading or writing by the NOS job.

If any PUTNLNK or PUTPLNK calls have been issued since the OPENLNK call, the CLOSLNK call writes an end-of-file delimiter on the link file. If the NOS/VE task is writing to the link file at the same time, a write deadlock occurs.

The subroutine call has the following format.

CALL CLOSLNK (status)

status

Name of 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.

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.

GETPLNK Subroutine

The GETPLNK subroutine reads a partial record of data from the link file.

The subroutine call has the following format.

CALL GETPLNK (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.

term

Name of the variable containing one of the following codes indicating which part of the record is being written:

1 Beginning of a record.

2 Continuation of a record.

3 End of a record.

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 call specified continuation of the record when the file position is not midrecord.

WREPLNK Subroutine

The WREPLNK subroutine writes a partition delimiter on the link file.

The subroutine call has the following format.

CALL WREPLNK (status)

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.

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 packed record format is required to access }
{ each character. Each character field must be }
{ described separately instead of as an array of }
{ 10 character fields because the CYBIL compiler }
{ byte-aligns any array whose total size is }
{ greater than 57 bits. (The array of characters }
{ would be 60 bits.) }
 word_array_type = ARRAY [1 .. num_words] OF
  PACKED RECORD
  unused_bits : 0 .. 15,
  char_1 : 0 .. 63,
  char_2 : 0 .. 63,
  char_3 : 0 .. 63,
  char_4 : 0 .. 63,
  char_5 : 0 .. 63,
  char_6 : 0 .. 63,
  char_7 : 0 .. 63,
  char_8 : 0 .. 63,
  char_9 : 0 .. 63,
  char_10 : 0 .. 63,
 RECEND;

 VAR
 output_file: [STATIC] amt$local_file_name :=
  '$OUTPUT',
 output_fid: amt$file_identifier;

{ The following procedure converts each character }
{ code in a string from display code to the }
{ corresponding character code in the ASCII character }
{ set. However, the display code for colon (00) is }
{ converted to the ASCII character code for a }
{ space (20). This is because the array read from }
{ the link file is zero-filled. When the array is }
{ converted, the zeros would be interpreted as colon }
{ codes and the string would have trailing colons. }
{ Because the colon is converted to blanks, the }
{ string has trailing blanks. }

PROCEDURE convert_display_code_to_ascii
 (display_code: word_array_type;
 VAR ascii_string: string(*));

 VAR
 string_position : 0 .. num_words*10,
 word_position : 1 .. num_words,
 char_position : 1 .. 10,
```

```
{ 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]);
```

```
   =6=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_6]);
   =7=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_7]);
   =8=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_8]);
   =9=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_9]);
   =10=
   ascii_string(string_position) :=
   $CHAR(ascii[display_code[word_position].char_10]);
   CASEND;
  FOREND /char_loop/;
FOREND /word_loop/;

PROCEND convert_display_code_to_ascii;

{ The following procedure writes a string to the }
{ output file. }

PROCEDURE print_string (str: string(*));

  VAR
  working_storage_area: ^cell,
  working_storage_area_length:
    amt$working_storage_length,
  byte_address: amt$file_byte_address,
  status: ost$status;

  working_storage_area := ^str;
  working_storage_area_length := STRLENGTH(str);
  amp$put_next (output_fid, working_storage_area,
    working_storage_area_length, byte_address, status);
  IF NOT status.NORMAL THEN
    pmp$exit(status);
  IFEND;

PROCEND print_string;
```

```
{ 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);
```

```
  amp$open(output_file, amc$record, NIL, output_fid,
   status);

  print_string(' Output file opened.');

  clp$create_variable('NOS_JOB_RECORD',clc$string_value,
   osc$max_string_size, 1, 1, variable_scope,
   nos_job_record, status);
  IF NOT status.NORMAL THEN
   pmp$exit (status);
  IFEND;

  job_record.sv[1].size := 47;
{ This statement assigns the contents of the command }
{ variable. }
  job_record.sv[1].value :=
   'myjob,t10.;get,procfil.;begin,vewrite,procfil.';
  variable_value.kind := clc$string_value;
  variable_value.max_string_size := osc$max_string_size;
  variable_value.string_value := ^job_record.cv;

  print_string(' Command variable created.');

  clp$write_variable ('NOS_JOB_RECORD', variable_value,
   status);
  IF NOT status.NORMAL THEN
   pmp$exit(status);
  IFEND;

  print_string(' Command variable written. ');

{ The following call starts the NOS job. }
  amp$open(link_file, amc$record, ^link_access_selections,
   link_fid, status);
  IF NOT status.NORMAL THEN
   pmp$exit(status);
  IFEND;

  print_string(' Link file opened.');

{ This loop is exited when the NOS job has opened the }
{ link file. }
  REPEAT
   amp$fetch_access_information (link_fid, access_info,
    status);
   IF NOT status.NORMAL THEN
    pmp$exit (status);
   IFEND;
  UNTIL (access_info [1].last_op_status = amc$complete);

  print_string('NOS job returned AMC$COMPLETE status.');
```

```
{ 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;
```

When the NOS_READ program opens the link file, the NOS job defined in the variable NOS_JOB_RECORD is executed. It gets and executes a NOS procedure file named PROCFIL. The following is a listing of PROCFIL.

```
.PROC,VEWRITE.
GET,VEWRITE.
FTN5,I=VEWRITE.
GET,DATAFL.
LIBRARY,NVELIB/A.
LGO.
DAYFILE,L=LOG.
REPLACE,LOG.
EXIT.
DAYFILE,L=LOG.
REPLACE,LOG.
```

The NOS procedure gets and compiles the file containing the FORTRAN program (VEWRITE). It also gets the data file (DATAFL). The LIBRARY command assigns the file containing the link subroutines (NVELIB) to the library set. The dayfile for the NOS job is stored as file LOG.

The following is a source listing of the FORTRAN 5 program on file VEWRITE.

```
      PROGRAM VEWRITE (INPUT,OUTPUT,DATAFL,TAPE1=DATAFL)
      INTEGER STATUS, MESS, LEN, UNUSED, POS
      CHARACTER WSA(25)*10

      N=0
      I=1
C READY TO READ DATAFL

   10 READ(1,100,END=20) WSA(I)
  100 FORMAT (A10)
      I = I+1
      N=N+1
      GO TO 10
C DATAFL READ

   20 CONTINUE
      CALL OPENLNK (STATUS)
      IF (STATUS .NE. 0) GO TO 40
C LINK FILE OPENED
```

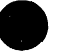
```
      CALL GETNLNK(MESS, 1, LEN, UNUSED, POS, STATUS)
      IF (STATUS .NE. 0) GO TO 40

      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 Variables 8-1
Variable Kind and Dimension 8-1
Variable Scope 8-2
CLP$CREATE_VARIABLE 8-3
CLP$DELETE_VARIABLE 8-5
CLP$READ_VARIABLE 8-6
CLP$WRITE_VARIABLE 8-8

String Conversion Procedures 8-10
CLP$CONVERT_INTEGER_TO_STRING 8-11
CLP$CONVERT_INTEGER_TO_RJSTRING 8-13
CLP$CONVERT_STRING_TO_INTEGER 8-15
CLP$CONVERT_STRING_TO_NAME 8-16
CLP$CONVERT_STRING_TO_FILE 8-17
CLP$CONVERT_VALUE_TO_STRING 8-18

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

Variable Scope

Each variable has a scope associated with it. The scope of a variable is the set of all blocks within which the variable can be accessed. The variable is discarded when processing leaves its scope.

The following are the possible scopes of a command language variable.

- Local: exists only in the current block.
- Job: exists in the job block. A job variable is retained across procedures and tasks in the job.
- XDCL: exists in the block that declares it and also in a subordinate block that declares the same variable as an XREF variable.
- XREF: exists in the block that declares it and in the surrounding block that declared the variable as an XDCL variable. The XREF declaration must exactly match the XDCL declaration.
- Utility: exists in the utility block. A utility variable is retained across utility subcommands.

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).

scope: clt$variable_scope;

Scope of declaration of the variable.

CLC$LOCAL_VARIABLE
Local variable.

CLC$JOB_VARIABLE
Job variable.

CLC$XDCL_VARIABLE
External declaration.

CLC$XREF_VARIABLE
External reference.

CLC$UTILITY_VARIABLE
Utility variable.

variable: VAR of clt$variable_reference;

Variable record [(a description of the created variable which can be used in subsequent CLP$WRITE_VARIABLE requests).actual variable storage].

status: VAR of ost$status;

Status record.

Condition Identifiers

cle$improper_var_declaration
cle$improper_variable_name
cle$unknown_utility
cle$var_already_declared

Remarks

- If its kind is string, the variable must have a maximum string length.
- The dimension of a variable is the upper and lower bounds of its subscripts. The upper bound minus the lower bound plus one is the number of elements in the variable. If the variable has only one element, its upper and lower bounds are the same (for example, 1..1 or 2..2).

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.

CLP$READ_VARIABLE

Purpose Returns a variable record.

Format **CLP$READ_VARIABLE (reference, variable, status)**

Parameters **reference**: string (*);

Variable name as specified on the CLP$CREATE_ VARIABLE call that created the variable. To reference a field, a subscript must be included to reference an element and a field name.

variable: VAR of clt$variable_reference;

Variable record as described in table 8-1.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Table 8-1. Variable Reference (CLT$VARIABLE_REFERENCE)

Field	Content
reference	Variable reference record (OST$STRING) **Field** / **Content** size: Actual string length (OST$STRING_SIZE, 0 through OSC$MAX_ STRING_SIZE) value: String (256 characters)
lower_bound	Lower array bound (CLT$VARIABLE_DIMENSION)
upper_bound	Upper array bound (CLT$VARIABLE_DIMENSION)
value	Variable value or values (CLT$VARIABLE_VALUE as described in table 8-2)

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.

CLP$WRITE_VARIABLE

Purpose Writes the value of a variable or a field or element of a variable.

Format **CLP$WRITE_VARIABLE (reference, value, status)**

Parameters **reference**: string (*);

Variable name as specified on the CLP$CREATE_ VARIABLE call that created the variable. To reference a field, a subscript must be included to reference an element and a field name.

value: clt$variable_value;

Record defining the value written in the variable as described in table 8-2.

status: VAR of ost$status;

Status record.

Condition Identifiers
cle$improper_variable_reference
cle$improper_variable_value
cle$undeclared_variable

Remarks The variable value is specified in a type CLT$VARIABLE_ VALUE record. If the variable is a string variable, the string value is specified as a pointer to an array of cell, although to be initialized, it must also be referenced as a string. For example, the following statements declare and initialize the string value ' This is the string value'. (The #UNCHECKED_ CONVERSION procedure is described in the CYBIL Language Definition manual. It copies the contents of a field directly without type conversion.)

```
VAR
  variable_value: clt$variable_value,
  x: record
    size: ost$string_size,
    value: string(256),
  recend,
  y: ^array[1..*] of cell;

x.size := 25;
x.value := ' This is the string value';
PUSH y: [1..#SIZE(x)];
#UNCHECKED_CONVERSION (x,y^);
```

(Continued)

(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;
```

String Conversion Procedures

The CLP$CONVERT_VALUE_TO_STRING converts a value returned by a CLP$GET_VALUE call (type CLT$VALUE) to a string.

The following procedures convert strings to other types.

CLP$CONVERT_STRING_TO_INTEGER
Converts a string to an integer.

CLP$CONVERT_STRING_TO_NAME
Converts a string to a name.

CLP$CONVERT_STRING_TO_FILE
Converts a file reference string to a local file name.

A reverse conversion may also be required. To display an integer, the program must convert the integer to its string representation. The following procedures convert integers to strings.

CLP$CONVERT_INTEGER_TO_STRING
Converts an integer to a left-justified string.

CLP$CONVERT_INTEGER_TO_RJSTRING
Converts an integer to a right-justified string.

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.

Condition Identifier None.

Remarks

- If requested, a trailing radix enclosed in parentheses is included in the string.
- If the integer is negative, a minus sign is included as the leftmost character in the string.
- If the specified radix is greater than ten and the leftmost digit of the result is greater than nine, a leading zero digit is added. For example, if the integer value is 240 (decimal), the radix is 16, and the allocated string is three characters, the string representation is 0F0.

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.

Condition Identifier cle$string_too_short

Remarks

- If requested, a trailing radix enclosed in parentheses is included in the string.
- If the integer is negative, a minus sign is included in the string. Its position within the string depends on the fill character used. If the fill character is a space, the minus sign is positioned immediately before the first digit. If the fill character is not a space, the minus sign is the leftmost character in the string.
- If the specified radix is greater than ten and the leftmost digit of the result is greater than nine, a leading zero digit is added if space for the digit is available in the allocated string. For example, if the integer value is 240 (decimal), the radix specified is 16, and the string allocated is three characters, the string representation is 0F0.

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_NAME

Purpose Converts the string representation of a name to a name. It performs the following operations.

- Converts all lowercase letters to uppercase letters.
- Left-justifies the name within a 31-character string and pads the string with spaces.

Format **CLP$CONVERT_STRING_TO_NAME (str, name, status)**

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

String.

name: VAR of clt$name;

Record returned describing the name.

Field	Content
size	Actual name length within the value string (1 through 31).
value	Name string (31 characters).

status: VAR of ost$status;

Status record.

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

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.

CLP$CONVERT_VALUE_TO_STRING

Purpose

Converts a value described in a CLT$VALUE record to its string representation. (The CLP$GET_VALUE call returns a CLT$VALUE record.)

Format

CLP$CONVERT_VALUE_TO_STRING (value, str, status)

Parameters

value: clt$value;

Value (type CLT$VALUE described in table 9-1).

str: VAR of ost$string;

String record.

Field	Content
size	Actual string length left-justified within the value string (0 through 256).
value	String representation (256 characters).

status: VAR of ost$status;

Status record.

Condition Identifier

None.

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 Processor 9-1
Parameter Descriptor Table (PDT) 9-2
PDT Declaration Syntax 9-2
Application Value Scanner 9-4
Retrieving Parameter List Information 9-11
CLP$SCAN_PARAMETER_LIST 9-12
CLP$TEST_PARAMETER 9-13
CLP$GET_SET_COUNT 9-14
CLP$GET_VALUE_COUNT 9-15
CLP$TEST_RANGE 9-16
CLP$GET_VALUE 9-17
CLP$GET_PARAMETER 9-18
CLP$GET_PARAMETER_LIST 9-19
File References 9-20
CLP$GET_PATH_DESCRIPTION 9-21
CLP$GET_WORKING_CATALOG 9-24
CLP$SET_WORKING_CATALOG 9-25
Subparameter Lists 9-26
CLP$PUSH_PARAMETERS 9-27
CLP$POP_PARAMETERS 9-28

Command Utility 9-29
Utility Command List Search Mode 9-30
CLP$PUSH_UTILITY 9-31
CLP$POP_UTILITY 9-35
Utility Subcommands 9-36
CLP$SCAN_COMMAND_FILE 9-37
CLP$END_SCAN_COMMAND_FILE 9-38
Command Utility Example 9-38

Utility Functions 9-52
CLP$SCAN_ARGUMENT_LIST 9-57

Token Scanning 9-58
CLP$SCAN_TOKEN 9-61

Expression Evaluation 9-62
CLP$SCAN_EXPRESSION 9-63

Command File Input 9-64
CLP$COLLECT_COMMANDS 9-65
CLP$GET_COMMAND_ORIGIN 9-66
CLP$GET_DATA_LINE 9-67
CLP$SCAN_COMMAND_LINE 9-68

Scanning Declarations 9-69
CLP$SCAN_PROC_DECLARATION 9-71
PDT Pointers 9-73
Parameter Descriptor 9-75
Value Kind Specifier 9-77

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.

Parameter Descriptor Table (PDT)

A PDT lists all parameter names and a parameter descriptor for each parameter. If CLP$SCAN_PARAMETER_LIST finds a parameter name undefined in the PDT or a parameter specification not allowed by the corresponding parameter descriptor, it returns a standard SCL syntax error in its status variable. The command processor can pass the syntax error to the user that entered the command by returning the status via the status variable passed to the command processor.

PDT Declaration Syntax

With a few exceptions, a PDT declaration uses the same syntax as an SCL procedure header. The syntax is almost the same because the SCL interpreter calls CLP$SCAN_PROC_DECLARATION to parse the PDT or procedure header input.

A PDT declaration differs from an SCL procedure header as follows:

- The first word of the declaration is PDT, instead of PROC.
- Only one name can be specified for the PDT; a procedure declaration can specify multiple procedure names.
- Any expression used within the PDT declaration must be a valid CYBIL expression because the expression is evaluated by the CYBIL compiler, not the SCL interpreter. Such an expression can include spaces only within parentheses or quotes.

The general format of an SCL procedure header is as follows:

PROC procedure_names *(parameter definitions)*

The general format of a PDT declaration is as follows:

PDT pdt_variable_name (parameter definitions)

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.

Application Value Scanner

If a parameter value to be defined within a PDT or an SCL procedure header is not to be evaluated as one of the system-supplied parameter value kinds, you can define your own parameter value kind.

The system-supplied parameter value kinds are FILE, NAME, STRING, INTEGER, BOOLEAN, and STATUS. When defining your own parameter value kind, you specify a name of your own choosing for the value kind. The name is used in the value specification in the parameter definition for the parameter.

The SCL interpreter can return the parameter value as an unevaluated string or call a procedure you define to evaluate the parameter value. If the SCL interpreter is not to evaluate the parameter value, you must specify the name of the procedure to perform the evaluation after the value kind name in the value specification. For example, if the value kind name is COMPLEX and the procedure to evaluate the value is COMPLEX_SCAN, the following is a parameter definition for a parameter named NUMBER using the value kind.

```
NUMBER: COMPLEX COMPLEX_SCAN
```

If you omit the second name, the parameter value is returned as a string with no evaluation performed.

The user-defined value kind is called an application value, and the procedure that performs the evaluation is called an application value scanner. If an application value scanner is specified in a PDT declaration, the application value scanner procedure must be declared within the program. If the application value scanner is specified within an SCL procedure header, the SCL interpreter must be able to load the application value scanner when it processes the SCL procedure.

When the CLP$SCAN_PARAMETER_LIST or CLP$SCAN_EXPRESSION procedure calls an application value scanner, it passes to the scanner the value name and keyword list from the value kind specifier and the string to be evaluated. The scanner returns a status record and a CLT$VALUE record containing the evaluated expression.

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)

Field	Content
descriptor	Name of the value kind returned. The CLP$CONVERT_VALUE_TO_STRING procedure returns the descriptor name for an application value if the field is not blank.
kind	Kind of value returned. CLC$UNKNOWN_VALUE Unknown value kind. CLC$APPLICATION_VALUE User-defined value kind. The application field contains the value. CLC$VARIABLE_REFERENCE Reference to a command variable. The var_ref field contains the variable reference. CLC$STRING_VALUE String value. The str field contains the string record. CLC$FILE_VALUE Local file name. The file field contains the file record. CLC$NAME_VALUE Name. The name field contains the name record. CLC$REAL_VALUE This value is currently unimplemented. CLC$INTEGER_VALUE Integer value. The int field contains the integer record. CLC$BOOLEAN_VALUE Boolean value. The bool field contains the boolean record. CLC$STATUS_VALUE Status record. The status field contains the status record.

(Continued)

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

Field	Content
int	Integer record (CLT$INTEGER). This field is generated only if the kind field is set to CLC$INTEGER_VALUE.
	Field / **Content**
	value: Integer value (integer).
	radix: Radix used (2 through 16).
	radix_specified: Indicates whether a radix was specified. TRUE Radix specified. FALSE Radix omitted.
bool	Boolean record (CLT$BOOLEAN). This field is generated only if the kind field is set to CLC$BOOLEAN_VALUE.
	Field / **Content**
	value: Boolean value (boolean).
	kind: Indicates keyword used to specify value (CLT$BOOLEAN_KINDS). CLC$TRUE_FALSE_BOOLEAN Value TRUE or FALSE. CLC$YES_NO_BOOLEAN Value YES or NO. CLC$ON_OFF_BOOLEAN Value ON or OFF.
status	Status record (OST$STATUS). This field is generated only if the kind field is set to CLC$STATUS_VALUE.

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

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

(Continued)

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

Field	Content
max_string_size	Maximum string size (OST$STRING_SIZE, 0 through OSC$MAX_STRING_SIZE, 256).
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.

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$SCAN_PARAMETER_LIST

Purpose Scans a parameter list.

Format **CLP$SCAN_PARAMETER_LIST (parameter_list, pdt, status)**

Parameters **parameter_list**: clt$parameter_list;

Parameter list (adaptable sequence).

pdt: clt$parameter_descriptor_table;

Parameter descriptor table.

status: VAR of ost$status;

Status record.

Condition Identifiers Any command language condition whose code is within the range 170100 through 170699 or 171000 through 171099.

Remarks CLP$SCAN_PARAMETER_LIST parses the parameter list according to the parameter definitions within the specified PDT. It checks that all parameter names within the parameter list are defined in the PDT and that each parameter value is valid for its parameter. If it finds an invalid parameter name or parameter value, it returns a standard SCL syntax error in the status variable.

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_SET_COUNT

Purpose Returns the number of value sets supplied for a parameter.

Format **CLP$GET_SET_COUNT (parameter_name, value_set_count, status)**

Parameters

parameter_name: string (*);

Parameter name.

value_set_count: VAR of 0 .. clc$max_value_sets;

Number of value sets.

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_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$TEST_RANGE

Purpose Determines whether a value is specified as a range.

Format **CLP$TEST_RANGE (parameter_name, value_set_number, value_number, range_specified, status)**

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

Parameter name.

value_set_number: 1 .. clc$max_value_sets;

Value set number.

value_number: 1 .. clc$max_values_per_set;

Value number.

range_specified: VAR of boolean;

Indicates whether the value is a range.

TRUE

The value is specified as a range.

FALSE

The value is not specified as a range.

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

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

Purpose Returns the entire value list for a parameter.

Format **CLP$GET_PARAMETER (parameter_name, value_list, status)**

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

Parameter name.

value_list: VAR of ost$string;

Value list record.

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

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_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.

File References

When the SCL interpreter evaluates a file reference expression, it assigns a local file name and validation ring for the file. The command processor can then access the file via its local file name.

However, the local file name record does not describe the catalog path for the file or the current working catalog for the job. If the command processor requires this additional information about the file reference, it can call the following procedures.

CLP$GET_PATH_DESCRIPTION

Returns each component of a file reference.

CLP$GET_WORKING CATALOG

Returns the current working catalog.

If the current working catalog is not appropriate, the program can change it with the following procedure.

CLP$SET_WORKING_CATALOG

Changes the current working catalog.

For more information on catalog paths and working catalogs, see the SCL Language Definition manual.

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).

path_container: VAR of clt$path_container;

Path storage area.

path: VAR of ^pft$path;

Pointer to the path storage area. The path storage area (type PFT$PATH) is an array of one or more PFT$NAME entries. Each entry contains a catalog or file name.

cycle_selector: VAR of clt$cycle_selector;

File cycle (see table 9-4).

open_position: VAR of clt$open_position;

Open position record having the following fields.

specified

Key field indicating whether the open position was specified.

TRUE

Open position is specified in the value field.

FALSE

Open position is not specified.

value

Positioning of the file when it is opened (AMT$OPEN_POSITION).

AMC$OPEN_NO_POSITIONING

No positioning.

AMC$OPEN_AT_BOI

Position at the beginning-of-information.

AMC$OPEN_AT_BOP

Position at the beginning-of-partition.

AMC$OPEN_AT_EOI

Position at the end-of-information.

status: VAR of ost$status;

Status record.

Condition Identifier

None.

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$GET_WORKING_CATALOG

Purpose Returns the description of the current working catalog.

Format **CLP$GET_WORKING_CATALOG (catalog_reference, path_container, path, status)**

Parameters **catalog_reference**: VAR of clt$file_reference;

Absolute path.

Field	Content
path_name	Absolute path name (CLT$PATH_NAME, 256-character string).
path_name_size	Actual length of the path name (1..CLC$MAX_PATH_NAME_SIZE, 256).
validation_ring	Indicates whether the ring is known and, if so, provides the ring number (see table 9-3).

path_container: VAR of clt$path_container;

Path storage area.

path: VAR of ^pft$path;

Pointer to the path storage area. The path storage area (type PFT$PATH) is an array of one or more PFT$NAME entries. Each entry contains a catalog name.

status: VAR of ost$status;

Status record.

Condition Identifier None.

Remarks The working catalog is the default catalog used if no catalog is specified in a file reference. The initial working catalog within 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$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.

Subparameter Lists

You can call the SCL interpreter to scan a parameter list that is not specified as input to a command. The parameter list could be input specifications within a file. For example, the SCU source library conversion commands use an input file in which each line is scanned as a parameter list. Each line must use SCL parameter syntax in the form.

OLD_NAME=name NEW_NAME=name

Or a parameter list could be a parameter specified in a parameter list. The parameter string could be scanned as a subparameter list.

An application value scanner can process a parameter string as a subparameter list. To do so, it performs the following steps.

1. Establishes a new parsing environment by calling the CLP$PUSH_PARAMETERS procedure.
2. Calls the CLP$SCAN_PARAMETER_LIST procedure to parse the parameter string. The call must specify a PDT for the subparameter list.
3. Gets and interprets the results of the parsing operation.
4. Calls CLP$POP_PARAMETERS to return to the previous parsing environment.

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.

CLP$POP_PARAMETERS

Purpose Returns to the previous parameter list environment.

Format **CLP$POP_PARAMETERS (status)**

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

Status record.

Condition Identifier cle$unexpected_call_to

Remarks CLP$POP_PARAMETERS is called after a CLP$PUSH_PARAMETERS call has been issued and the parameter list has been scanned.

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.

Utility Command List Search Mode

The command list search mode for a utility determines whether the global command list is searched. The global command list is a stack of pointers. Each pointer points to a command environment that includes a command list. When a command environment is established, its pointer is pushed on the stack. When a command environment is disestablished, its pointer is popped from the stack.

The possible command list search modes are global, restricted, and exclusive. If the utility search mode is global and the SCL interpreter does not find the command in the utility command list, the interpreter searches the command lists of previous entries on the stack. If you specify restricted or exclusive search modes, the utility can restrict the search to the utility command list.

If the utility search mode is not exclusive, entry of a command in escape mode is allowed. Entry of a command in escape mode is indicated by a \ command prefix. In escape mode, the SCL interpreter skips the utility command list and begins its search for the command at the next entry in the stack.

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.

commands: ^clt$sub_command_list;

Utility subcommands list pointer. It points to an adaptable array of one or more CLT$SUB_COMMAND_LIST_ENTRY records. Each record has the following fields.

name

Subcommand name (OST$NAME, 31-character string).

kind

Key field (CLT$SUB_COMMAND_LIST_ENTRY_KIND).

CLC$LINKED_SUB_COMMAND

The command processor is already loaded and linked within the task address space. The subcommand pointer is in the command field.

CLC$UNLINKED_SUB_COMMAND

The command processor is not yet loaded or linked. The subcommand entry point is in the procedure_name field.

CLC$PROCEDURE_SUB_COMMAND

The subcommand processor is an SCL procedure or a program description module to be started as a separate task. The subcommand SCL procedure name is in the procedure_name field.

command

Pointer to subcommand procedure (type CLT$COMMAND).

procedure_name

Name of a CYBIL entry point or an object library module (type PMT$PROGRAM_NAME).

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.

Remarks

- The command environment includes storage for command language variables and a subcommand list and function list for the utility. CLP$PUSH_UTILITY pushes the subcommand and function lists on the task's SCL command stack so that the SCL interpreter recognizes utility subcommands and functions.

- The identifier specified in the kind field of the commands parameter variable depends on the characteristics of the command processor.

  - CLC$LINKED_SUB_COMMAND is the most commonly used command processor kind. The command processor is loaded with the task.

  - CLC$UNLINKED_SUB_COMMAND should be used when the command processor is large and is executed infrequently. By using the CLC$UNLINKED_SUBCOMMAND kind, the command processor is loaded only when its command is executed, thereby saving space. To find the command processor, the system searches the entry point directories of the object libraries in the program library list. For more information, see the PMP$LOAD description.

  - CLC$PROCEDURE_SUB_COMMAND is used when the command processor is an SCL procedure or a program to be executed as a separate task. As for CLC$UNLINKED_SUBCOMMAND, the system searches the entry point directories of the object libraries in the program library list to find the command processor.

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

Utility Subcommands

A utility subcommand is a command defined in the utility subcommand list. The subcommand list specifies a command processor for each subcommand. As described in the CLP$PUSH_UTILITY description, it contains either a CYBIL procedure pointer or an entry point name to be found in the program library list.

When the command utility calls CLP$PUSH_UTILITY, the utility subcommand list specified on the call becomes the top command list on the SCL command stack for the task. After the command utility calls CLP$SCAN_COMMAND_FILE, the SCL interpreter begins reading commands from the command file. Because the utility command list is the first list on the command stack, the SCL interpreter first searches the utility command list for each command entry.

If the command is in the utility subcommand list, the SCL interpreter calls the command processor specified for the subcommand. It passes the command parameter list and a status variable to the command processor as described under Command Processor in this chapter.

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.

CLP$END_SCAN_COMMAND_FILE

Purpose Ends the interpretation of command input from the command file.

Format **CLP$END_SCAN_COMMAND_FILE (utility_name, status)**

Parameters **utility_name**: ost$name;

Utility name specified on the preceding CLP$SCAN_COMMAND_FILE call.

status: VAR of ost$status;

Status record.

Condition Identifier cle$unknown_utility

Command Utility Example

This section lists the CYBIL source statements for a command utility named INFO_PLEASE. The format of the INFO_PLEASE command is as follows:

INFO_PLEASE
OUTPUT=file
STATUS=status variable

OUTPUT (O)

File on which utility output is written. If OUTPUT is omitted, the utility output is written on file $OUTPUT.

STATUS

Optional status variable.

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.
/
```

The following is a listing of the source program for the INFO_PLEASE command utility.

```
MODULE command_utility_example;

*copyc clp$scan_parameter_list
*copyc pmp$exit
*copyc clp$get_value
*copyc amp$open
*copyc amp$put_next
*copyc clp$push_utility
*copyc clp$scan_command_file
*copyc amp$close
*copyc clp$pop_utility
*copyc clp$end_scan_command_file
*copyc clp$convert_integer_to_string
*copyc pmp$get_task_cp_time
*copyc pmp$get_srus
*copyc pmp$get_processor_attributes
*copyc pmp$get_account_project

{ The command name must be an entry point in the }
{ program; therefore, the following program name is }
{ the command name, info_please. }

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

```
{ 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;
```

```
{ The following five procedures are the command }
{ processors for the utility commands.  All five }
{ procedures have the same general form.  Each }
{ procedure declaration must specify a parameter }
{ list variable and a status variable although none }
{ of the utility commands use parameters.  Each }
{ procedure calls a program service procedure to }
{ return a record of information.  If necessary, the }
{ procedure converts the information to strings.  It }
{ inserts the information in a message displayed by }
{ the put_message procedure. }

{ This procedure processes the processor command. }
{ The command returns the processor model number,}
{ serial number, and page size in bytes. }

PROCEDURE processor_command
  (processor_parameter_list: clt$parameter_list;
  VAR stat: ost$status);

  VAR
    attributes: pmt$processor_attributes,
    serial_no_string: ost$string,
    page_size_string: ost$string,
    message_ptr_1, message_ptr_2: ^string( * );

pmp$get_processor_attributes(attributes,stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

put_message(' Processor attributes:');

CASE attributes.model_number OF
= pmc$cpu_model_p1 =
  put_message ('   CPU model P1');
= pmc$cpu_model_p2 =
  put_message ('   CPU model P2');
= pmc$cpu_model_p3 =
  put_message ('   CPU model P3');
CASEND;

clp$convert_integer_to_string(attributes.serial_number,
  10, false, serial_no_string, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;
```

```
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 srus command.  It }
{ returns the number of SRUs accumulated by the job. }

PROCEDURE srus_command
  (srus_parameter_list: clt$parameter_list;
  VAR stat: ost$status);

  VAR
    srus: jmt$sru_count,
    srus_string: ost$string,
    message_ptr: ^string( * );

pmp$get_srus(srus, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

clp$convert_integer_to_string(srus, 10, false,
  srus_string, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

PUSH message_ptr: [srus_string.size+7];
message_ptr^(1,1) := ' ';
message_ptr^(2,srus_string.size) :=
  srus_string.value(2, srus_string.size);
message_ptr^((srus_string.size + 1),6) := ' SRUs.';
put_message (message_ptr^);

PROCEND srus_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;
```

```
PUSH message_ptr_1: [job_mode_string.size+28];
message_ptr_1^(1,3) := '   ';
message_ptr_1^(4,job_mode_string.size) :=
  job_mode_string.value(1,job_mode_string.size);
message_ptr_1^((job_mode_string.size+4),25) :=
  ' microseconds in job mode';
put_message(message_ptr_1^);

clp$convert_integer_to_string(cp_time.monitor_time,
  10, false, monitor_mode_string, stat);
IF NOT stat.normal THEN
  pmp$exit(stat);
IFEND;

PUSH message_ptr_2: [monitor_mode_string.size+32];
message_ptr_2^(1,3) := '   ';
message_ptr_2^(4,monitor_mode_string.size) :=
  monitor_mode_string.value(2,monitor_mode_string.size);
message_ptr_2^((monitor_mode_string.size+4),29) :=
  ' microseconds in monitor mode';
put_message(message_ptr_2^);

PROCEND cp_time_command;
```

```
{ 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;
```

```
{ The main program begins here. }
VAR
  output_file: clt$value,
  utility_name: [STATIC] ost$name :=
    'INFO_PLEASE_UTILITY',
  output_fid: amt$file_identifier,
  sub_command_list_ptr: ^clt$sub_command_list;

{ The following statements declare and initialize }
{ the PDT for the info_please command.  It defines }
{ each parameter in the command parameter list. }

pdt info_please_pdt (
output,o: file = $output; status)

?? PUSH (LISTEXT := ON) ??

VAR
  info_please_pdt: [STATIC, READ, cls$pdt]
    clt$parameter_descriptor_table :=
    [^info_please_pdt_names, ^info_please_pdt_params];

VAR
  info_please_pdt_names: [STATIC, READ,
  cls$pdt_names_and_defaults] array [1 .. 3] of
    clt$parameter_name_descriptor := [
    ['OUTPUT', 1],
    ['O', 1],
    ['STATUS', 2]];

VAR
  info_please_pdt_params: [STATIC, READ,
    cls$pdt_parameters] array [1 .. 2] of
    clt$parameter_descriptor := [

{ OUTPUT O }
    [[clc$optional_with_default, ^info_please_pdt_dv1],
    1, 1,
    1, 1,
    clc$value_range_not_allowed,
    [NIL,
    clc$file_value]],
```

```
{ 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 statements initialize the utility }
{ command list. }

PUSH sub_command_list_ptr: [1..5];
sub_command_list_ptr^[1].name := 'processor';
sub_command_list_ptr^[1].kind :=
  clc$linked_sub_command;
sub_command_list_ptr^[1].command := ^processor_command;
sub_command_list_ptr^[2].name := 'srus';
sub_command_list_ptr^[2].kind :=
  clc$linked_sub_command;
sub_command_list_ptr^[2].command := ^srus_command;
sub_command_list_ptr^[3].name := 'cp_time';
sub_command_list_ptr^[3].kind :=
  clc$linked_sub_command;
sub_command_list_ptr^[3].command := ^cp_time_command;
sub_command_list_ptr^[4].name := 'acct_proj';
sub_command_list_ptr^[4].kind :=
  clc$linked_sub_command;
sub_command_list_ptr^[4].command := ^acct_proj_command;
sub_command_list_ptr^[5].name := 'quit';
sub_command_list_ptr^[5].kind :=
  clc$linked_sub_command;
sub_command_list_ptr^[5].command := ^quit_command;

{ The following call pushes the utility command list }
{ on the SCL command stack.  When the SCL }
{ interpreter scans the command file, it searches }
{ the utility command list first for the commands. }

clp$push_utility (utility_name,
  clc$global_command_search, sub_command_list_ptr,NIL,
  status);
IF NOT status.normal THEN
  pmp$exit(status);
IFEND;
```

```
{ 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;
```

Utility Functions

A utility function is a function that can be called within a parameter expression in the utility parameter list. All utility functions are listed in the utility function list.

The utility function list is specified as an array pointer on the CLP$PUSH_UTILITY call. Each entry of the array contains a function name and references the function processor for the named function.

A function processor is passed the following parameters.

```
procedure (function_name: clt$name;
           argument_list: string( * );
           VAR value: clt$value;
           VAR status: ost$status)
```

function_name

Function name that called the procedure.

argument_list

Actual argument list specified for the function.

value

Value returned by the function. The CLT$VALUE type is described in table 9-1.

status

Status record.

The function processor can call the CLP$SCAN_ARGUMENT_LIST procedure to parse its argument list. It parses the argument_list string it received to the CLP$SCAN_ARGUMENT_LIST procedure with an argument descriptor table (ADT) and an argument value table (AVT).

The ADT guides the parsing of the argument list. It is an adaptable array of one or more argument descriptors (CLT$ARGUMENT_DESCRIPTOR_TABLE). See table 9-5 for the structure of the array elements.

The CLP$SCAN_ARGUMENT_LIST procedure returns the actual argument values in an AVT. The AVT must be an adaptable array of one or more value records (type CLT$ARGUMENT_VALUE_TABLE). The structure of the value records is shown in table 9-1 (CLT$VALUE).

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)

Field	Content
keyword_ values	Pointer to the array of valid keywords (adaptable array of OST$NAME). Each keyword must be in name format (31 characters, left-justified, with blank fill). The array must list all valid keywords for the parameter. If the parameter value cannot be a keyword, the pointer must be NIL.
kind	Key field indicating the valid value kind for the evaluated expression (CLT$VALUE_KINDS). CLC$ANY_VALUE Any value kind. CLC$KEYWORD_VALUE Keyword listed in the keyword_values array. CLC$VARIABLE_REFERENCE Command variable. The variable_kind field specifies the variable kind, and the array_allowed field specifies whether the variable can be an array. CLC$APPLICATION_VALUE Application value. The scanner field specifies the application value scanner, and the value_name field specifies the name passed to the scanner. CLC$FILE_VALUE File or catalog reference. CLC$NAME_VALUE Name. The min_name_size field specifies the minimum name length, and the max_name_size field specifies the maximum name length. CLC$STRING_VALUE String. The min_string_size field specifies the minimum string length, and the max_string_size field specifies the maximum string length. CLC$INTEGER_VALUE Integer. The min_integer_value field specifies the minumum integer value, and the max_integer_value field specifies the maximum integer value.

(Continued)

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)

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

Field	Content
scanner	Application value scanner (see Application Value Scanner in this chapter) (used only if the kind is CLC$APPLICATION_VALUE). **Field** / **Content** kind — Key field indicating scanner kind (CLT$AV_SCANNER_KIND). CLC$UNSPECIFIED_AV_SCANNER No application value scanner is specified for the value. The expression is stored as a string record (OST$STRING) in the application field of the CLT$VALUE record returned. CLC$LINKED_AV_SCANNER The application value scanner is specified in the proc field. CLC$UNLINKED_AV_SCANNER The application value scanner is specified in the name field. proc — Pointer to the scanner procedure (CLT$APPLICATION_VALUE_SCANNER). name — Name of scanner module (PMT$PROGRAM_NAME).
min_name_size	Minimum name length in characters (1 through 31). It is used only if kind is CLC$NAME_VALUE.
max_name_size	Maximum name length in characters (1 through 31). It is used only if kind is CLC$NAME_VALUE.
min_string_size	Minimum string length in characters (1 through 256). It is used only if kind is CLC$STRING_VALUE.
max_string_size	Maximum string length in characters (1 through 256). It is used only if kind is CLC$STRING_VALUE.
min_integer_size	Minimum integer size (integer). It is used only if kind is CLC$INTEGER_VALUE.
max_integer_size	Maximum integer size (integer). It is used only if kind is CLC$INTEGER_VALUE.

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.

Token Scanning

A CLP$SCAN_TOKEN call returns the next token in a string. The call passes the string and the string index where the scan begins. The procedure returns the token found and the string index where the next scan can begin.

The token is returned in a record of type CLT$TOKEN (see table 9-7).

The token kinds and their definition in BNF notation are listed in table 9-8.

A name token is left-justified; its size cannot exceed 31 characters. Lowercase characters are converted to uppercase characters.

An integer token can have a radix; if a radix is omitted, decimal (base 10) is assumed.

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

Field	Content
text_index	String index where the token begins (OST$STRING_INDEX, 1, through OSC$MAX_STRING_SIZE+1).
text_size	Number of characters in the token (OST$STRING_SIZE, 0, through OSC$MAX_STRING_SIZE, 256).
descriptor	String describing the token type; the string can be used in an error message.
kind	Key field identifying the token kind (CLT$LEXICAL_KINDS). CLC$UNKNOWN_TOKEN Unknown token stored in str field. CLC$STRING_TOKEN String stored in str field. CLC$NAME_TOKEN Name stored in name field. CLC$INTEGER_TOKEN Integer stored in int field. CLC$REAL_TOKEN Floating point number stored in rnum field.

(Continued)

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

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

Table 9-8. Token Definitions

Tokens that must be delimited:

name ::= <alphabetic char> [<alphanumeric char>] ...
<alphanumeric char> ::= <alphabetic char> | <digit>
<alphabetic char> ::= <letter> | . | $ | # | @

integer ::= <digit> [<hex digit>] ... [<(> <radix> <)>]

real ::= <mantissa> [<exponent>]
<mantissa> ::= <integer part> <.> <fraction part>
<integer part> ::= <unsigned decimal>
<fraction part> ::= <unsigned decimal>
<exponent> ::= <E | e> [<+ | ->] <unsigned decimal>
<unsigned decimal> ::= <digit> ...

string ::= ' [<string char>] ... '
<string char> ::= <any ASCII character except '> | ''

Delimiter tokens:

comma ::= [<sp>] , [<sp>]
assignment ::= [<sp>] = [<sp>]
semicolon ::= [<sp>] ; [<sp>]
reverse slant ::= [<sp>] \ [<sp>]
ellipsis ::= [<sp>] .. [.] ... [<sp>]
left parenthesis ::= ([<sp>]
left bracket ::= [[<sp>]
left brace ::= { [<sp>]

query ::= ? [<sp>]
right parenthesis ::= [<sp>])

right bracket ::= [<sp>]]
right brace ::= [<sp>] }

dot ::= .
colon ::= :
exponentiation ::= [<sp>] ** [<sp>]
multiplication ::= [<sp>] * [<sp>]
division ::= [<sp>] / [<sp>]
concatenation ::= [<sp>] // [<sp>]
greater than ::= [<sp>] > [<sp>]
greater than or equal ::= [<sp>] >= [<sp>]
less than ::= [<sp>] < [<sp>]
less than or equal ::= [<sp>] <= [<sp>]
equal ::= [<sp>] = [<sp>]
not equal ::= [<sp>] <> [<sp>]
addition ::= [<sp>] + [<sp>]
subtraction ::= [<sp>]–[<sp>]

<sp> := < > | HT | <comment>

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.

Expression Evaluation

The CLP$SCAN_EXPRESSION procedure can evaluate a string. A call to the procedure passes the expression string and a value kind specifier that guides the evaluation. The procedure returns the value in a record of type CLT$VALUE.

Value kind specifiers are described in table 9-6. The CLT$VALUE type is described in table 9-1.

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.

Command File Input

The SCL interpreter reads the command stream for the job from the command file. The primary command file for a job is called COMMAND (CLC$JOB_COMMAND_INPUT). The file from which the SCL interpreter is currently reading commands is called $COMMAND (CLC$CURRENT_COMMAND_INPUT).

The following calls direct the SCL interpreter to perform the following actions.

CLP$COLLECT_COMMANDS

Copies commands read from the command file to another file without interpreting or processing the commands.

CLP$GET_COMMAND_ORIGIN

Returns the origin of a command (either interactive or batch origin).

CLP$GET_DATA_LINE

Returns the next line read from the command file without interpreting or processing the line.

CLP$SCAN_COMMAND_LINE

Interprets the line specified on the call as it would interpret a line read from the command file.

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_COMMAND_ORIGIN

Purpose Determines whether the command is from an interactive terminal.

Format **CLP$GET_COMMAND_ORIGIN (interactive, status)**

Parameters **interactive**: VAR of boolean;

Indicates whether the command is from an interactive device.

TRUE

Interactive command.

FALSE

Not an interactive command.

status: VAR of ost$status;

Status record.

Condition Identifier None.

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.

CLP$SCAN_COMMAND_LINE

Purpose Interprets a line as a command entry.

Format **CLP$SCAN_COMMAND_LINE (text, status)**

Parameters **text**: string (*);
Command line.

status: VAR of ost$status;
Status record.

Condition Identifier Any command language condition.

Remarks

- CLP$SCAN_COMMAND_LINE directs the SCL interpreter to interpret the line specified on the call as it would interpret a line read from the command file.
- If a command interpreted within the line does not specify a status variable, its completion status is stored in the status variable specified on the CLP$SCAN_COMMAND_LINE call.

  For example, suppose the following line is scanned.

  ```
  'DISPLAY_CATALOG OOPS STATUS=STAT;DISPLAY_LOG'
  ```

  The completion status of the DISPLAY_CATALOG command is returned in the STAT variable. The completion status of the DISPLAY_LOG command is returned in the status variable specified on the CLP$SCAN_COMMAND_LINE call. Because the DISPLAY_CATALOG command has its own status variable, the DISPLAY_LOG command is interpreted even if the DISPLAY_CATALOG command returns abnormal status.

  Contrast the preceding command line with the following command line.

  ```
  'DISPLAY_CATALOG OOPS;DISPLAY_LOG'
  ```

  Because the DISPLAY_CATALOG command does not specify a status variable, the completion status of the DISPLAY_CATALOG command is returned in the status variable specified on the CLP$SCAN_COMMAND_LINE call. If the DISPLAY_CATALOG command returns abnormal status, the DISPLAY_LOG command is not interpreted.

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.

The following is an example of a preprocessing procedure that returns one line of input to CLP$SCAN_PROC_DECLARATION.

```
PROCEDURE get_pdt_line (VAR line: ost$string;
  VAR index: ost$string_index;
  VAR token: clt$token;
  VAR status: ost$status);
  VAR got_line: boolean;

status.normal := TRUE;
REPEAT
  index := 1;
  line.size := 0;
{ The get_line procedure determines whether the }
{ line is a continuation line and, if so, discards }
{ the ellipsis (..) and concatenates the line with }
{ the next line.  It repeats the concatenation }
{ process until it reads a line that is not a }
{ continuation line.  It then returns the full line. }


  get_line (line, got_line, status);
  IF NOT (status.normal AND got_line) THEN
    RETURN;
  IFEND;

{ The first CLP$SCAN_TOKEN call determines whether }
{ the line contains only spaces or comments.  If it }
{ does, the call returns CLC$EOL_TOKEN in the }
{ token.kind field indicating the end of the line. }
{ If it returns CLC$SPACE_TOKEN, CLP$SCAN_TOKEN is }
{ called a second time to return the second token }
{ in the line.  If the first token is }
{ CLC$SPACE_TOKEN, it consists of all leading }
{ spaces and comments on the line.  The token }
{ returned by the preprocessing procedure must be }
{ the first meaningful token on the line. }

  clp$scan_token (line.value (1, line.size), index,
    token, status);
  IF status.normal AND (token.kind = clc$space_token)
    THEN
    clp$scan_token (line.value (1, line.size), index,
      token, status);
  IFEND;
UNTIL (NOT status.normal) OR
  (token.kind < > clc$eol_token);

PROCEND get_pdt_line;
```

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.)

extra_info_area: VAR of SEQ (*);

Adaptable sequence in which the procedure CLP$SCAN_PROC_DECLARATION stores the additional parameter descriptor information such as keyword arrays and default specifications. The parameter descriptor in the parameter_area sequence points to its information in the extra_info_area sequence.

proc_names: VAR of ^clt$proc_names;

Pointer to an array containing all procedure names. (If the input_type is CLC$PDT_INPUT, only one name is stored.)

pdt: VAR of clt$parameter_descriptor_table;

Parameter descriptor table (PDT).

symbolic_parameters: VAR of ^clt$symbolic_parameters;

Pointer to an array of CLT$SYMBOLIC_PARAMETER records. Each record contains the following pointers to the expression strings specified within the parameter value specification:

min_value_sets

Minimum value sets [^string(*)].

max_value_sets

Maximum value sets [^string(*)].

min_value_per_set

Minimum values per value set [^string(*)].

max_value_per_set

Maximum values per value set [^string(*)].

value_kind_qualifier_low

Lower bound [^string(*)].

value_kind_qualifier_high

Upper bound [^string(*)].

status: VAR of ost$status;

Status record.

Condition Identifiers

Any command language condition whose code is within the range 170100 through 170599 or 170800 through 171099.

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)

Field	Content
names	Pointer to an array listing all parameter names (^array [1..*] of CLT$PARAMETER_NAME_DESCRIPTOR). The order of the names in the array is irrelevant except when an error in a positional parameter is reported. The error is reported using the name in that position of the names array. **Field** / **Content** name: Name used to specify the parameter when it is specified by name, rather than position (type OST$NAME). number: Index into the parameter descriptors array of the entry describing the parameter (1 through CLC$MAX_PARAMETERS). More than one name can reference the same parameter descriptor.
parameters	Pointer to an array listing the parameter descriptors (^ array [1 .. *] of CLT$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.

SCL parameters can be specified by keyword or by position within the parameter list. If a parameter is specified by keyword, NOS/VE searches the list of parameter keywords for the specified keyword. If it finds the keyword, it uses the parameter descriptor referenced by the keyword entry to parse the parameter.

While parsing the parameter list, NOS/VE keeps a pointer to the last parameter descriptor used. Then, whenever a parameter is specified by position, NOS/VE uses the next parameter descriptor in the parameter descriptor list to parse the parameter. If all parameters are specified by position, the parameter descriptors are used in order beginning with the first entry.

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.

Table 9-10. Parameter Descriptor (CLT$PARAMETER_DESCRIPTOR)

Field	Content
required_or_optional	Indicates whether the parameter is required or optional and its default, if any (CLT$REQUIRED_OR_OPTIONAL). **Field** / **Content** selector — Key field determining whether the parameter is required or optional. CLC$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 contains a pointer to the default value. default — Pointer to the default value (^string(*)).
min_value_sets	Minimum number of value sets for the parameter (1 .. CLC$MAX_VALUE_SETS).
max_value_sets	Maximum number of value sets for the parameter (1 .. CLC$MAX_VALUE_SETS).
min_values_per_set	Minimum number of values per value set (1 .. CLC$MAX_VALUES_PER_SET).
max_values_per_set	Maximum number of values per value set (1 .. CLC$MAX_VALUES_PER_SET).
value_range_allowed	Indicates whether a parameter value can be specified as a range of values. CLC$VALUE_RANGE_NOT_ALLOWED A value cannot be specified as a range. CLC$VALUE_RANGE_ALLOWED A value can be specified as a range.
value_kind_specifier	Indicates the kind of value allowed for this parameter (CLT$VALUE_KIND_SPECIFIER, see table 9-6).

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-1

ASCII Character Set B-1

Constant and Type Declarations C-1

Stack Frame Save Area D-1

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.

D

Deck

A sequence of lines in a source library that can be manipulated as a unit by the Source Code Utility (SCU).

E

End-of-Information (EOI)

The point at which data in the file ends.

Exception Condition

A situation that, when detected by a procedure caller, indicates an abnormal completion of the called procedure.

Execution Ring

The level of hardware protection assigned to a procedure while it is executing.

External Reference

A call to an entry point in another module.

F

FAP

A File Access Procedure.

File Attribute

A characteristic of a file. Each file has a set of attributes that completely define file structure and processing limitations.

File Reference

An SCL element that identifies a file and, optionally, the file position to be established prior to use. The format of a file reference is as follows:

file.file position

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.

P

Page

An allocatable unit of real memory.

Parameter List

A series of parameters separated by spaces or commas.

Parameter Value

See Value.

Partition

A unit of data on a sequential or byte addressable file delimited by end-of-partition separators or the beginning- or end-of-information.

Path

Identifies a file. It may include the family name, user name, subcatalog name or names, and file name.

Pointer

The virtual address of a value.

Program Attribute

A characteristic of a program as defined in the program description or by a job default value.

Program Description

Information that defines a program, including the modules that comprise the program and the options to be used when it is executed.

Program Library List

A list of object libraries searched for modules during the loading of a program.

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.

S

SCL Procedure

A sequence of SCL commands executed when the procedure name is entered. It can be stored as a module on an object library.

Segment

One or more pages assigned to a file. The segment has the ring attributes of the file.

Starting Procedure

The entry point at which program execution is to begin.

System Resource Units (SRUs)

An accounting unit used to measure resource usage by a job or task.

T

Task

The instance of execution of a program.

Token

A character unit meaningful to SCL categorized by the parsing of a string.

Transfer Symbol

An entry point within a module at which program execution can begin.

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.

Value Set

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

value element or (value element,value element,value element)

See Value, Value Element, and Value List.

Value Set Count

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

Virtual Address

An address of a location in virtual memory. The system hardware determines the physical location referenced by the virtual address.

Virtual Memory

The memory access mode using virtual addresses. Each task uses the same virtual address range; the system associates the virtual address range referenced by a task to the physical memory assigned to the task.

W

Working Catalog

The catalog used if no other catalog is specified on a file reference.

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

ASCII Code			Graphic or Mnemonic	Name or Meaning
Decimal	Hexadecimal	Octal		
000	00	000	NUL	Null
001	01	001	SOH	Start of heading
002	02	002	STX	Start of text
003	03	003	ETX	End of text
004	04	004	EOT	End of transmission
005	05	005	ENQ	Enquiry
006	06	006	ACK	Acknowledge
007	07	007	BEL	Bell
008	08	010	BS	Backspace
009	09	011	HT	Horizontal tabulation
010	0A	012	LF	Line feed
011	0B	013	VT	Vertical tabulation
012	0C	014	FF	Form feed
013	0D	015	CR	Carriage return
014	0E	016	SO	Shift out
015	0F	017	SI	Shift in
016	10	020	DLE	Data link escape
017	11	021	DC1	Device control 1
018	12	022	DC2	Device control 2
019	13	023	DC3	Device control 3
020	14	024	DC4	Device control 4
021	15	025	NAK	Negative acknowledge
022	16	026	SYN	Synchronous idle
023	17	027	ETB	End of transmission block
024	18	030	CAN	Cancel
025	19	031	EM	End of medium
026	1A	032	SUB	Substitute
027	1B	033	ESC	Escape
028	1C	034	FS	File separator
029	1D	035	GS	Group separator
030	1E	036	RS	Record separator
031	1F	037	US	Unit separator
032	20	040	SP	Space
033	21	041	!	Exclamation point
034	22	042	"	Quotation marks
035	23	043	#	Number sign
036	24	044	$	Dollar sign
037	25	045	%	Percent sign
038	26	046	&	Ampersand
039	27	047	'	Apostrophe
040	28	050	(	Opening parenthesis
041	29	051	)	Closing parenthesis
042	2A	052	*	Asterisk
043	2B	053	+	Plus
044	2C	054	,	Comma
045	2D	055	-	Hyphen
046	2E	056	.	Period
047	2F	057	/	Slant

(Continued)

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

ASCII Code			Graphic or	
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)

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

ASCII Code				
Decimal	Hexadecimal	Octal	Graphic or Mnemonic	Name or Meaning
092	5C	134	\	Reverse slant
093	5D	135	]	Closing bracket
094	5E	136	^	Circumflex
095	5F	137	_	Underline
096	60	140	`	Grave accent
097	61	141	a	Lowercase a
098	62	142	b	Lowercase b
099	63	143	c	Lowercase c
100	64	144	d	Lowercase d
101	65	145	e	Lowercase e
102	66	146	f	Lowercase f
103	67	147	g	Lowercase g
104	68	150	h	Lowercase h
105	69	151	i	Lowercase i
106	6A	152	j	Lowercase j
107	6B	153	k	Lowercase k
108	6C	154	l	Lowercase l
109	6D	155	m	Lowercase m
110	6E	156	n	Lowercase n
111	6F	157	o	Lowercase o
112	70	160	p	Lowercase p
113	71	161	q	Lowercase q
114	72	162	r	Lowercase r
115	73	163	s	Lowercase s
116	74	164	t	Lowercase t
117	75	165	u	Lowercase u
118	76	166	v	Lowercase v
119	77	167	w	Lowercase w
120	78	170	x	Lowercase x
121	79	171	y	Lowercase y
122	7A	172	z	Lowercase z
123	7B	173	{	Opening brace
124	7C	174	|	Vertical line
125	7D	175	}	Closing brace
126	7E	176	~	Tilde
127	7F	177	DEL	Delete

Constant and Type Declarations C

This appendix lists the constant and type declarations used by the procedures described in this manual. In general, the declarations are listed in alphabetical order by identifier name. However, the numeric order of ordinal constants is maintained.

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                           ';
```

```
  clc$max_arguments = osc$max_string_size DIV 2;
  clc$max_keyword_values = 7fffffff(16);
  clc$max_parameter_names = 7fffffff(16);
  clc$max_parameters = 7fffffff(16);
  clc$max_path_elements = clc$max_path_name_size DIV 2;
  clc$max_path_name_size = osc$max_string_size;
  clc$max_proc_names = osc$max_string_size DIV 2;
  clc$max_significant_digits = 28;
  clc$max_value_sets = 7fffffff(16);
  clc$max_values_per_set = 7fffffff(16);
  clc$max_variable_dimension = 7fffffff(16);
  clc$min_variable_dimension = - 7fffffff(16);
  clc$null_file =
    '$NULL                       ';
  clc$proc_caller_command_input =
    '$COMMAND_OF_CALLER          ';
  clc$standard_input =
    '$INPUT                      ';
  clc$standard_output =
    '$OUTPUT                     ';

SECTION
  cls$adt: READ;
  cls$adt_names_and_defaults: READ;

SECTION
  cls$pdt: READ;
  cls$pdt_parameters: READ;
  cls$pdt_names_and_defaults: READ;
```

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$boolean = record
  value: boolean,
  kind: clt$boolean_kinds,
recend;

clt$boolean_kinds = (clc$true_false_boolean,
  clc$yes_no_boolean, clc$on_off_boolean);

clt$command = ^procedure (parameter_list:
  clt$parameter_list;
  VAR status: ost$status);

clt$command_search_modes =
  (clc$global_command_search,
  clc$restricted_command_search,
  clc$exclusive_command_search);

clt$cycle_selector = record
  specification: clt$cycle_specification,
  value: pft$cycle_selector,
recend;

clt$cycle_specification = (clc$cycle_omitted,
  clc$cycle_specified, clc$cycle_next_highest,
  clc$cycle_next_lowest);

clt$file = record
  local_file_name: amt$local_file_name,
recend;

clt$file_reference = record
  path_name: clt$path_name,
  path_name_size: 1 .. clc$max_path_name_size,
  validation_ring: record
    case known: boolean of
    = TRUE =
      number: ost$valid_ring,
    casend,
  recend,
recend;

clt$function = ^procedure (function_name: clt$name;
  argument_list: string ( * );
  VAR value: clt$value;
  VAR status: ost$status);
```

```
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$open_position = record
  case specified: boolean of
  = TRUE =
    value: amt$open_position,
  = FALSE =
    ,
  casend,
recend;

clt$parameter_descriptor = record
  required_or_optional: clt$required_or_optional,
  min_value_sets: 1 .. clc$max_value_sets,
  max_value_sets: 1 .. clc$max_value_sets,
  min_values_per_set: 1 .. clc$max_values_per_set,
  max_values_per_set: 1 .. clc$max_values_per_set,
  value_range_allowed: (clc$value_range_not_allowed,
    clc$value_range_allowed),
  value_kind_specifier: clt$value_kind_specifier,
recend;

clt$parameter_descriptor_table = record
  names: ^array [1 .. * ] of
  clt$parameter_name_descriptor,
  parameters: ^array [1 .. * ] of
    clt$parameter_descriptor,
recend;

clt$parameter_list = pmt$program_parameters;

clt$parameter_list_contents = ost$string;

clt$parameter_name_descriptor = record
  name: ost$name,
  number: 1 .. clc$max_parameters,
recend;

clt$path_container = SEQ (REP clc$max_path_elements
  of pft$name);

clt$path_name = string (clc$max_path_name_size);

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

```
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;
```

```
clt$sub_command_list_entry_kind =
  (clc$linked_sub_command, clc$unlinked_sub_command,
  clc$procedure_sub_command);

clt$symbolic_parameters = array [1 .. * ] of
  clt$symbolic_parameter;

clt$symbolic_parameter = record
  min_value_sets: ^string ( * ),
  max_value_sets: ^string ( * ),
  min_values_per_set: ^string ( * ),
  max_values_per_set: ^string ( * ),
  value_kind_qualifier_low: ^string ( * ),
  value_kind_qualifier_high: ^string ( * ),
recend;

clt$token = record
  text_index: ost$string_index,
  text_size: ost$string_size,
  descriptor: string (osc$max_name_size),
  case kind: clt$lexical_kinds of
  = clc$unknown_token .. clc$string_token =
    str: ost$string,
  = clc$name_token =
    name: clt$name,
  = clc$integer_token =
    int: clt$integer,
  = clc$real_token =
    rnum: clt$real,
  casend,
recend;

clt$value = record
  descriptor: string (osc$max_name_size),
  case kind: clc$unknown_value .. clc$status_value of
  = clc$unknown_value =
    ,
  = clc$application_value =
    application: clt$application_value,
  = clc$variable_reference =
    var_ref: clt$variable_reference,
  = clc$string_value =
    str: ost$string,
  = clc$file_value =
    file: clt$file,
  = clc$name_value =
    name: clt$name,
```

```
  = 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;
```

```
    clt$value_kinds = (clc$unknown_value,
      clc$application_value, clc$variable_reference,
      clc$file_value, clc$name_value, clc$string_value,
      clc$real_value, clc$integer_value,
      clc$boolean_value, clc$status_value,
      clc$any_value);

    clt$variable_dimension =
      clc$min_variable_dimension ..
      clc$max_variable_dimension;

    clt$variable_kinds = clc$string_value ..
      clc$status_value;

    clt$variable_reference = record
      reference: ost$string,
      lower_bound: clt$variable_dimension,
      upper_bound: clt$variable_dimension,
      value: clt$variable_value,
    recend;

    clt$variable_value = record
      descriptor: string (osc$max_name_size),
      case kind: clt$variable_kinds of
      = clc$string_value =
{ The max_string_size and string_value fields which }
{ follow should be interpreted as though they were }
{ replaced by: }
{       string_value: ^array [1 .. * ] of record
{         current_string_size: ost$string_size,
{         value: string ( * ),
{       recend,
{ where STRLENGTH(string_value^[i].value) =
{   max_string_size and }
{   string_value^[i].current_string_size <=
{   max_string_size }
        max_string_size: ost$string_size,
        string_value: ^array [1 .. * ] of cell,
      = clc$real_value =
        real_value: ^array [1 .. * ] of clt$real,
      = clc$integer_value =
        integer_value: ^array [1 .. * ] of clt$integer,
      = clc$boolean_value =
        boolean_value: ^array [1 .. * ] of clt$boolean,
      = clc$status_value =
```

```
{ 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);
```

IF

Constants

```
ifc$interactive_facility_id = 'IF';
ifc$pause_break = 1;
ifc$terminate_break = 2;
ifc$terminal_connection_broken = 3;
ifc$job_reconnect = 4;
```

Types

```
ift$interactive_condition = pmt$condition_identifier;
```

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;
```

MM

Constants

```
mmc$memory_management_id = 'MM';
mmc$sac_read_beyond_eoi  = 1;
mmc$sac_read_write_beyond_msl  = 2;
mmc$sac_segment_access_error = 3;
mmc$sac_key_lock_violation = 4;
mmc$sac_ring_violation = 5;
mmc$sac_io_read_error  = 6;
mmc$sac_no_append_permission = 7;
mmc$sac_tape_system_failure = 8;
```

Types

```
mmt$segment_access_condition = record
  identifier: pmt$condition_identifier,
  segment: ^cell,
recend;
```

OF

Constants

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

Types

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

OS

Constants

```
    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$invalid_ring = 0;

    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$max_status_message_line = osc$max_string_size;
    osc$max_status_message_lines = 7fffffff(16);
    osc$max_string_size = 256;

    osc$maximum_offset = 7fffffff(16);
    osc$maximum_segment = 0fff(16);

    osc$min_page_size = 512;
    osc$min_ring = 1; { Lowest ring number (most }
{     privleged). }
    osc$min_status_message_line = 32;

    osc$null_name = '                              ';
    osc$operating_system_id = 'OS';
    osc$os_ring_1 = 1; { Reserved for Operating System. }
    osc$sj_ring_1 = 4; { Reserved for system job. }
    osc$sj_ring_2 = 5;
    osc$sj_ring_3 = 6;
    osc$status_parameter_delimiter = CHR (31) { Unit }
{     Separator} ;
    osc$tmtr_ring = 2; { Task Monitor. }
    osc$tsrv_ring = 3; { Task services. }
    osc$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;
```

Types

```
ost$activity = record
  case activity: ost$wait_activity OF
   =osc$await_time=
     milliseconds: 0 .. 0FFFFFFFF(16),
   =pmc$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$date_formats = (osc$default_date,
  osc$month_date, osc$mdy_date, osc$iso_date,
  osc$ordinal_date, osc$dmy_date);

ost$date_time = record
  year: 0..255,  {year minus 1900, e.g. 80 = 1980}
  month: 1..12,
  day: 1..31,
  hour: 0..23,
  minute: 0..59,
  second: 0..59,
  millisecond: 0..999,
recend;

ost$dmy_date = string (8);

ost$family_name = ost$name;

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;

ost$iso_date = string (10);

ost$key_lock = packed record
  global: boolean, { True if value is global key. }
  local: boolean, { True if value is local key. }
  value: ost$key_lock_value, { Key or lock value. }
recend;

ost$key_lock_value = 0 .. 3f(16);

ost$max_status_message_line =
  osc$min_status_message_line ..
    osc$max_status_message_line;

ost$mdy_date = string (8);
```

```
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$register_number = 0 .. 0f(16);

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$stack_frame_save_area = record
  minimum_save_area: ost$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;

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$status_message = SEQ
  (ost$status_message_line_count, REP 24 of
  ost$status_message_line_size,REP 24 * 80 of char);
```

```
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;
```

```
ost$user_condition = (osc$privileged_instruction,
  osc$unimplemented_instruction, osc$free_flag,
  osc$process_interval_timer, osc$inter_ring_pop,
  osc$critical_frame_flag, osc$keypoint,
  osc$divide_fault, osc$debug,
  osc$arithmetic_overflow, osc$exponent_overflow,
  osc$exponent_underflow, osc$fp_significance_loss,
  osc$fp_indefinite, osc$arithmetic_significance,
  osc$invalid_bdp_data);

ost$user_conditions = set OF ost$user_condition;

ost$user_identification = record
  user: ost$user_name,
  family: ost$family_name,
recend;

ost$user_name = ost$name;

ost$valid_ring = osc$min_ring ..
  osc$max_ring; { Valid Ring Number. }

ost$virtual_machine_identifier =
  (osc$cyber_180_mode, osc$cyber_170_mode,
  osc$50_reserved, osc$51_reserved, osc$52_reserved,
  osc$53_reserved, osc$54_reserved, osc$55_reserved,
  osc$56_reserved, osc$57_reserved, osc$58_reserved,
  osc$59_reserved, osc$60_reserved, osc$61_reserved,
  osc$62_reserved, osc$63_reserved);

ost$x_register = integer;

ost$wait = (osc$wait, osc$nowait);

ost$wait_activity = (ost$null_activity, osc$await_time,
  pmc$await_task_termination, pmc$await_local_queue_message);

ost$wait_list = array [1 .. *] of ost$activity;
```

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$change_list = array [ 1 .. * ] of
  pft$change_descriptor;

pft$change_type = (pfc$pf_name_change,
  pfc$password_change, pfc$cycle_number_change,
  pfc$retention_change, pfc$log_change,
  pfc$charge_change);

pft$cycle_number = pfc$minimum_cycle_number ..
  pfc$maximum_cycle_number;

pft$cycle_options = (pfc$lowest_cycle,
  pfc$highest_cycle, pfc$specific_cycle);

pft$cycle_selector = record
  case cycle_option: pft$cycle_options of
  = pfc$lowest_cycle =
    ,
  = pfc$highest_cycle =
    ,
  = pfc$specific_cycle =
    cycle_number: pft$cycle_number,
  casend,
recend;
```

```
pft$group = record
  case group_type: pft$group_types of
  = pfc$public =
    ,
  = pfc$family =
    family_description: record
      family: ost$family_name,
    recend,
  = pfc$account =
    account_description: record
      family: ost$family_name,
      account: avt$account_name,
    recend,
  = pfc$project =
    project_description: record
      family: ost$family_name,
      account: avt$account_name,
      project: avt$project_name,
    recend,
  = pfc$user =
    user_description: record
      family: ost$family_name,
      user: ost$user_name,
    recend,
  = pfc$user_account =
    user_account_description: record
      family: ost$family_name,
      account: avt$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;
```

```
pft$group_types = (pfc$public, pfc$family,
  pfc$account, pfc$project, pfc$user,
  pfc$user_account, pfc$member);

pft$log = (pfc$log, pfc$no_log);

pft$name = ost$name;

pft$password = pft$name;

pft$path = array [ 1 .. * ] of pft$name;

pft$permit_options = (pfc$read, pfc$shorten,
  pfc$append, pfc$modify, pfc$execute, pfc$cycle,
  pfc$control);

pft$permit_selections = set of pft$permit_options;

pft$retention = pfc$minimum_retention ..
  pfc$maximum_retention;

pft$share_options = pfc$read .. pfc$execute;

pft$share_requirements = set of pft$share_options;

pft$share_selections = set of pft$share_options;

pft$usage_options = pfc$read .. pfc$execute;

pft$usage_selections = set of pft$usage_options;

pft$wait = (pfc$wait, pfc$no_wait);
```

PM

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$condition = record
  case selector: pmt$condition_selector of
  = pmc$system_conditions =
    system_conditions: pmt$system_conditions,
    untranslatable_pointer: ost$pva,
  = pmc$block_exit_processing =
    reason: pmt$block_exit_reason,
  = jmc$job_resource_condition =
    job_resource_condition:
      jmt$job_resource_condition,
  = mmc$segment_access_condition =
    segment_access_condition:
      mmt$segment_access_condition,
  = ifc$interactive_condition =
    interactive_condition: ift$interactive_condition,
  = pmc$pit_condition =
    ,
  = pmc$user_defined_condition =
    user_condition_name: pmt$condition_name,
  = pmc$condition_combination =
    combination: pmt$condition_combination,
  casend,
recend;

pmt$condition_combination = set of
  pmc$system_conditions ..
  pmc$user_defined_condition;

pmt$condition_handler = ^procedure
  (condition {input} : pmt$condition;
  condition_descriptor {input} :
  ^pmt$condition_information;
  save_area {input, output} :
  ^ost$stack_frame_save_area;
  VAR status {output} : ost$status);

pmt$condition_handler_active = record
  system: pmt$system_conditions,
  segment_access: mmt$segment_access_condition,
recend;
```

```
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$load_map_option = (pmc$no_load_map,
  pmc$segment_map, pmc$block_map,
  pmc$entry_point_map, pmc$entry_point_xref);

pmt$load_map_options = set of pmt$load_map_option;

pmt$loaded_address = record
  case kind: pmt$loaded_address_kind of
  = pmc$procedure_address =
    pointer_to_procedure: ^procedure,
  = pmc$data_address =
    pointer_to_data: ^cell,
  casend,
recend;

pmt$loaded_address_kind = (pmc$procedure_address,
  (pmc$data_address);

pmt$local_binary_logs = pmc$job_statistic_log ..
  pmc$job_statistic_log;

pmt$local_binary_logset = set of
  pmt$local_binary_logs;

pmt$log_msg_origin = (pmc$msg_origin_command,
  pmc$msg_origin_system, pmc$msg_origin_program,
  pmc$msg_origin_command_skip,
  pmc$msg_origin_recovery);

pmt$log_msg_text = string ( * );

pmt$logs = (pmc$job_statistic_log, pmc$account_log,
  pmc$engineering_log, pmc$statistic_log,
  pmc$system_log, pmc$job_log);

pmt$logset = set of pmt$logs;

pmt$message = record
  sender_id: pmt$task_id, { set by pmp$send_to_queue }
  sender_ring: ost$ring, { set by pmp$send_to_queue }
  case contents: pmt$message_kind of
  = pmc$message_value =
    value: pmt$message_value,
  = pmc$passed_segments, pmc$shared_segments =
    { * not supported in R1 }
    number_of_segments: pmt$segments_per_message,
    segments: array [pmt$segments_per_message] of
      pmt$queued_segment,
  casend,
recend;
```

```
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$program_attributes = record
  contents: pmt$prog_description_contents,
  starting_procedure: pmt$program_name,
  number_of_object_files:pmt$number_of_object_files,
  number_of_modules: pmt$number_of_modules,
  number_of_libraries: pmt$number_of_libraries,
  load_map_file: amt$local_file_name,
  load_map_options: pmt$load_map_options,
  termination_error_level:
    pmt$termination_error_level,
  preset: pmt$initialization_value,
  maximum_stack_size: ost$segment_length,
  debug_input: amt$local_file_name,
  debug_output: amt$local_file_name,
  abort_file: amt$local_file_name,
  debug_mode: pmt$debug_mode,
recend;

pmt$program_description = SEQ ( * );

pmt$prog_description_content =
  (pmc$starting_proc_specified,
  pmc$object_file_list_specified,
  pmc$module_list_specified,
  pmc$library_list_specified,
  pmc$load_map_file_specified,
  pmc$load_map_options_specified,
  pmc$term_error_level_specified,
  pmc$preset_specified,
  pmc$max_stack_size_specified,
  pmc$debug_input_specified,
  pmc$debug_output_specified,
  pmc$abort_file_specified,
  pmc$debug_mode_specified, pmc$pd_reserved_11,
  pmc$pd_reserved_10, pmc$pd_reserved_9,
  pmc$pd_reserved_8, pmc$pd_reserved_7,
  pmc$pd_reserved_6, pmc$pd_reserved_5,
  pmc$pd_reserved_4, pmc$pd_reserved_3,
  pmc$pd_reserved_2, pmc$pd_reserved_1);
```

```
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);
```

```
pmt$system_condition=(pmc$detected_uncorrected_err,
  pmc$ua_unselectable, pmc$sw_unselectable,
  pmc$instruction_specification,
  pmc$address_specification, pmc$xr_unselectable,
  pmc$access_violation,
  pmc$environment_specification,
  pmc$xi_unselectable, pmc$pf_unselectable,
  pmc$sc_unselectable, pmc$sit_unselectable,
  pmc$invalid_segment_ring_0,
  pmc$out_call_in_return, pmc$sel_unselectable,
  pmc$tx_unselectable, pmc$privileged_instruction,
  pmc$unimplemented_instruction, pmc$ff_unselectable,
  pmc$pit_unselectable, pmc$inter_ring_pop,
  pmc$cff_unselectable, pmc$kypt_unselectable,
  pmc$divide_fault, pmc$debug_unselectable,
  pmc$arithmetic_overflow, pmc$exponent_overflow,
  pmc$exponent_underflow, pmc$fp_significance_loss,
  pmc$fp_indefinite, pmc$arithmetic_significance,
  pmc$invalid_bdp_data);

pmt$system_conditions = set of pmt$system_condition;

pmt$task_cp_time = record
  task_time: 0 .. 7ffffffffff(16),
  monitor_time: 0 .. 7ffffffffff(16),
recend;

pmt$task_id = 0 .. pmc$max_task_id;

pmt$task_status = record
  complete: boolean,
  status: ost$status,
recend;

pmt$termination_error_level =
  (pmc$warning_load_errors, pmc$error_load_errors,
  pmc$fatal_load_errors);

pmt$time_increment = record
  year: integer,
  month: integer,
  day: integer,
  hour: integer,
  minute: integer,
  second: integer,
  millisecond: integer,
recend;
```

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.

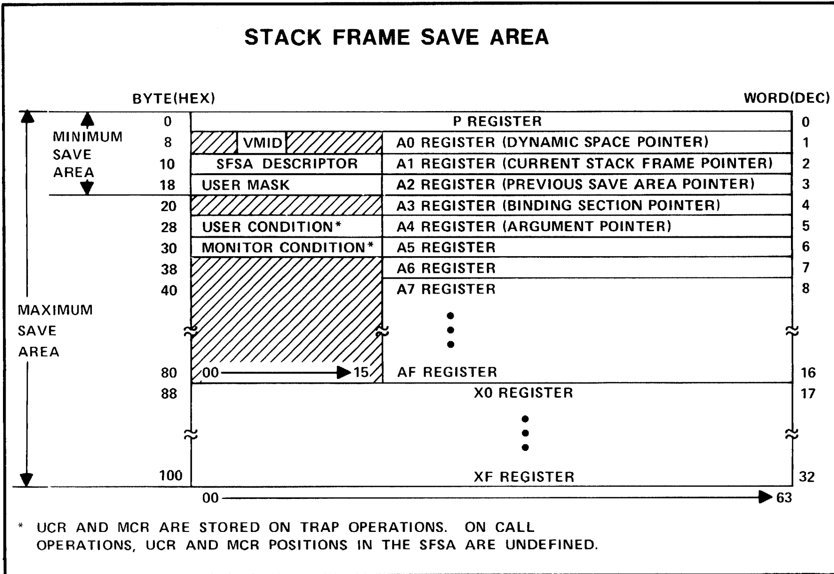


Figure D-1. Stack Frame Save Area Format

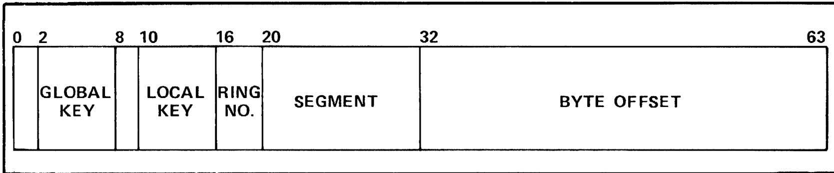


Figure D-2. P Register Format

```
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 OST$STACK_FRAME_SAVE_AREA)

minimum_save_area

Packed record containing the minimum stack frame information (type OST$MINIMUM_SAVE_AREA).

Field	Content
p_register	Packed record containing current program register (type OST$P_REGISTER). Fields and contents are as follows: undefined 0 through 3 hexadecimal. global_key Type OST$KEY_LOCK_VALUE, 0 through 3F hexadecimal. undefined 0 through 3 hexadecimal. local_key Type OST$KEY_LOCK_VALUE, 0 through 3F hexadecimal. pva Process virtual address (type OST$PVA, packed record). ring Ring number. seg Segment number. offset Byte offset.
vmid	One of the following virtual machine identifiers (type OST$VIRTUAL_MACHINE_IDENTIFIER): OSC$CYBER_180_MODE OSC$CYBER_170_MODE
undefined	0 through 0FFF hexadecimal.
a0_dynamic_space_pointer	Dynamic space pointer (^cell).

(Continued)

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-1. Stack Frame Save Area (Type OST$STACK_FRAME_SAVE_AREA *(Continued)*

undefined

0 through 0FFFF hexadecimal.

a3

Content of A3 register (^cell). The A registers can only contain pointers.

user_condition_register

Set of user conditions (type OST$USER_CONDITIONS, see table D-2)

a4

Content of A4 register (^cell).

monitor_condition_register

Set of monitor conditions (type OST$MONITOR_CONDITIONS, see table D-3).

a5

Content of A5 register (^cell).

a_registers

Contents of A registers (six- through sixteen-record array as follows by field and content):

Field	Content
undefined	0 through 0FFFF hexadecimal
a_register	Register value (^cell). An A register can only contain a pointer.
x_registers	Contents of X registers (type array OST$REGISTER_NUMBER] of OST$X_REGISTER). An X register can contain either a pointer or a value.

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.

Table D-3. Monitor Conditions (Type OST$MONITOR_CONDITIONS)

Following are the identifiers and meanings for the OST$MONITOR_CONDITIONS.

OSC$DETECTED_UNCORRECTED_ERR
Uncorrectable error detected.

OSC$NOT_ASSIGNED
This bit is not set explicitly by any condition.

OSC$SHORT_WARNING
Short warning environmental failure.

OSC$INSTRUCTION_SPEC
Improper instruction specification.

OSC$ADDRESS_SPECIFICATION
Improper address specification.

OSC$EXCHANGE_REQUEST
CYBER 170 mode exchange request.

OSC$ACCESS_VIOLATION
Memory access failed because of lack of access permission.

OSC$ENVIRONMENT_SPEC
Environment specification error.

OSC$EXTERNAL_INTERRUPT
Interrupt received from an external processor.

OSC$PAGE_FAULT
Page entry not found in page table.

OSC$SYSTEM_CALL
Exchange interrupt from job process state to monitor process state.

OSC$SYSTEM_INTERVAL_TIMER
System interval timer decremented to zero.

OSC$INVALID_SEGMENT_RING_0
Either an invalid segment number or ring number set to zero.

OSC$OUT_CALL_IN_RETURN
Processor attempted an outward call or an inward return.

OSC$SOFT_ERROR
Hardware malfunction detected and corrected.

OSC$TRAP_EXCEPTION
Fault detected during a trap interrupt operation.

Index

Index

A

A registers D-4
Abnormal status 1-8
Abort A-1
Abort file execution 3-17
Aborting a task 3-18
Access_mode file attribute 7-5
Access violation condition D-6
Account name retrieval 2-18
Activity completion wait 4-2
Address retrieval for externally declared procedure 3-13
Address space 3-1
Address specification error condition D-6
ADT 9-52
ALTERNATE_BASE parameter 1-4
AMP$OPEN exception conditions 7-6
Appending a status parameter 6-4
Appending an integer as a status parameter 6-5
Application value scanner 9-4
Argument descriptor table 9-52
Argument list 9-52
Argument value table 9-52
Arithmetic overflow condition D-5
Arithmetic significance condition D-5
Assigning link file attributes 7-5
Attributes 7-5
- Data conversion 7-7
- Deadlock 7-7
- Exception conditions 7-6

Audience 7
AV declarations C-1
AVT 9-52
Awaiting activity completion 4-2
Awaiting task termination 3-15

B

Base library 1-4
Batch mode 2-19
Beginning-of-information A-1
Block exit processing condition handler 5-16
BOI A-1

C

Calling a system interface procedure 1-6
Catalog A-1
Catalog name A-1
Catalog path 9-20
Causing a condition 5-22
Central processor
- Attribute retrieval 2-16
- Time retrieval 2-22

Checking the completion status 1-8
CL declarations C-1
Clearing a link file deadlock 7-7
Clock value 2-13
Closing the link file 7-12
CLOSLNK subroutine 7-12
CLP$COLLECT_COMMANDS procedure 9-65
CLP$CONVERT_INTEGER_TO_RJSTRING procedure 8-13
CLP$CONVERT_INTEGER_TO_STRING procedure 8-11
CLP$CONVERT_STRING_TO_FILE procedure 8-17
CLP$CONVERT_STRING_TO_INTEGER procedure 8-15
CLP$CONVERT_STRING_TO_NAME procedure 8-16
CLP$CONVERT_VALUE_TO_STRING procedure 8-18
CLP$CREATE_VARIABLE procedure 8-3
CLP$DELETE_VARIABLE procedure 8-5
CLP$END_SCAN_COMMAND_FILE procedure 9-38

CLP$GET_COMMAND_ORIGIN procedure 9-66
CLP$GET_DATA_LINE procedure 9-67
CLP$GET_PARAMETER procedure 9-18
CLP$GET_PARAMETER_LIST procedure 9-19
CLP$GET_PATH_DESCRIPTION procedure 9-21
CLP$GET_SET_COUNT procedure 9-14
CLP$GET_VALUE procedure 9-17
CLP$GET_VALUE_COUNT procedure 9-15
CLP$GET_WORKING_CATALOG procedure 9-24
CLP$POP_PARAMETERS procedure 9-28
CLP$POP_UTILITY procedure 9-35
CLP$PUSH_PARAMETERS procedure 9-27
CLP$PUSH_UTILITY procedure 9-31
CLP$READ_VARIABLE procedure 8-6
CLP$SCAN_ARGUMENT_LIST procedure 9-57
CLP$SCAN_COMMAND_FILE procedure 9-37
CLP$SCAN_EXPRESSION procedure 9-63
CLP$SCAN_PARAMETER_LIST procedure 9-12
CLP$SCAN_PROC_DECLARATION procedure 9-71
CLP$SCAN_TOKEN procedure 9-61
CLP$SET_WORKING_CATALOG procedure 9-25
CLP$TEST_PARAMETER procedure 9-13
CLP$TEST_RANGE procedure 9-16
CLP$WRITE_VARIABLE procedure 8-8
Collecting commands on a file 9-65
Command definition 9-1
Command file
  Input 9-64
  Scan 9-37
Command language
  Processing 9-1
  Services 8-1
  Variables 8-1
Command list 9-1
  Search mode 9-30
Command origin 9-66
Command prefix character 9-30
Command processor 9-1
Command stack 9-29
Command utility 9-29
  Example 9-39
  Glossary definition A-1
Command variable
  Creation 7-2
  Initialization 8-1
  Scope 8-2
Comment submission 10
Communication between a NOS/VE task and a NOS job 7-10
Compact format for date and time 2-1
COMPILE file 1-4
Completion status 1-8
Computing the date and time 2-7
Condition code 1-9
Condition handler
  Establishment 5-6
  Glossary definition A-1
  Procedure declaration 5-12
  Processing 5-14
  Scope 5-4
  Status record initialization 6-6
Condition handling 5-4
Condition identifier 1-9
Condition information 1-9
Condition processing 5-1
Condition set 5-6
Connecting a task to a queue 4-7
Connection identifier 4-4

Continuing a condition 5-15
Converting
  Integer to left-justified string 8-11
  Integer to right-justified string 8-13
  String to file reference 8-17
  String to integer 8-15
  String to name 8-16
  String values 8-10
  Value to string 8-18
*COPYC directive 1-2
Copying procedure declaration decks 1-4
Copyright information 2
CP time 2-22
CPU attribute retrieval 2-16
Creating a NOS job 7-1
Creating the command variable 7-2
  Procedure description 8-3
Critical_frame_flag D-3
CYBER 170 exchange request D-6
CYBIL 7
  Element types 1-10
  Manual set 8
  Procedure call 1-6
  Procedures 1-1

D

Data conversion for a link file 7-7
Data transmission to and from a link file 7-8
Date and time retrieval 2-1
  Example 2-10
Debug interrupt condition D-5
Debug mode 3-10
Deck A-2
DECK parameter 1-4
Defining
  Conditions 5-21
  Parameter value kinds 9-4
  Queues 4-5
  SCL commands 9-1
Deleting a command variable 8-5
Dependencies 3-14
Description format 1-12
Detected uncorrected error condition D-6
Disabling system condition detection 5-3
Disconnecting a task from a queue 4-8
Disestablishing a condition handler 5-11
Disestablishing a parsing environment 9-28
Displaying a job status message 2-31
Divide fault condition D-5
Dynamic loading 3-10
Dynamic space pointer D-3

E

Enabling system condition detection 5-2
End-of-information A-2
Ending a command file scan 9-38
Entering a message in the job log 2-28
Environment specification error condition D-6
EOI A-2
Error codes 1-9
Error_exit_name file attribute 7-5
Error message formatting 6-10
Error message templates 6-10
Escape mode 9-30
Establishing a condition handler 5-9
Establishing a parsing environment 9-27
Evaluating expressions 9-62
Evaluating file references 9-20
Example programs
  Command utility 9-39
  Date and time retrieval 2-10
  Interstate communications 7-17
  Queue communication 4-16
  Sense switch retrieval 2-27
  System interface use 1-2
Exception condition 1-9
Exchange request condition D-6

Exclusive command list search mode 9-30
Executing a task 3-11
Execution mode retrieval 2-19
Execution ring A-2
Execution time retrieval 2-22
Exiting a task 3-19
EXPAND_DECK command 1-4
Expanding a source program 1-4
Exponent overflow condition D-5
Exponent underflow condition D-5
Expression evaluation 9-62
External interrupt condition D-6
External reference A-2

F

Family name retrieval 2-24
FAP A-2
FAP attribute 7-5
Fetching information about a link file 7-9
File attribute
 Glossary definition A-2
 Setting 7-5
File interface
 Status record conventions 6-2
File_organization file attribute 7-5
File reference
 Content 9-21
Fill character 8-14
Floating point indefinite condition D-5
Floating point significance loss condition D-5
Formatting
 Date 2-8
 Error message 6-12
 Time 2-9
Frame descriptor D-3
Free flag condition D-5
Function processor 9-52

G

General waiting mechanism 4-1
 Command description 9-3
 Parameter descriptor tables 9-2
 Standard error messages 6-10
 Unique names 2-14
GETNLNK subroutine 7-13
GETPLNK subroutine 7-14
Getting
 Account and project names 2-8
 Address of externally declared procedure 3-13
 Command origin 9-66
 Current date and time
  In compact format 2-6
  In legible format 2-4
 Current date in legible format 2-2
 Data from the link file 7-13
 Data line 9-67
 Job information 2-17
 Job mode 2-19
 Job names 2-20
 Message level 6-14
 Microsecond clock value 2-13
 Number of queues defined 4-15
 Operating system version 2-15
 Parameter list 9-19
 Parameter value from parameter list 9-17
 Parameter value list from parameter list 9-18
 Processor attributes 2-16
 Program description 3-8
 Program description size 3-7
 Queue information 4-12
 Queue limits 4-13
 Sense switch settings 2-26
 SRUs 2-21
 Status severity 6-9
 System information 2-13
 Task CP time 2-22
 Task identifier 2-23
 User identification 2-24
 Value count 9-15
 Value set count 9-14
 Working catalog 9-24
Global command list 9-30
 Search mode 9-30

Global key D-3
Glossary definition A-2

H

How to use
System interface calls 1-1
This manual 7

I

ICE$ exception conditions 7-6
IDENTIFICATION procedure 2-24
IF declarations C-12
Information returned 1-9
Inhibiting system conditions 5-3
Initializing a command variable 8-1,3,8
Initializing a program description 3-6
Initiating a task 3-10
Input parameters 1-6
Input preprocessing procedure 9-69
Instruction specification error condition D-6
Interactive condition handler 5-17
Interactive mode 2-19
Interpreting a string as a file reference 8-17
Inter_ring_pop condition D-5
Interstate communications 7-1
Example 7-17
Invalid BDP data condition D-5
Invalid segment condition D-6

J

JM declarations C-13
Job A-3
Job information retrieval 2-17
Job library list 3-5
Glossary definition A-3
Job local queues 4-4
Job log
Entry format 2-29
Messages 2-28
Job mode retrieval 2-19
Job mode time retrieval 2-22
Job names retrieval 2-20
Job resource condition handler 5-17
Job status message 2-30
Job variable 8-2

K

Keypoint condition D-5

L

Legible format for date and time 2-1
Lexical unit 9-61
Link file 7-1
Link subroutines 7-11
Linked function 9-33
Linked subcommand 9-34
List A-3
Local file name 8-17
Local key D-3
Local queues 4-4
Local variable 8-2
Log messages 2-28

M

Managing sense switches 2-25
Manual
Conventions 9
History 3
Organization 8
Mass storage backup error handling 5-18
Memory access violation condition D-6
Message formatting 6-10
Message generation 6-1
Message levels 6-10
Message sending via a queue 4-4
Message templates 6-10
Microsecond clock 2-13

Minimum_save_area D-3
MM declarations C-14
Model number retrieval 2-16
Module list 3-5
Monitor conditions D-6
Monitor time retrieval 2-22

N

Name generation 2-14
Naming convention 1-10
Network Operating System/ Virtual Environment (see NOS/VE)
Normal status 1-8
NOS link subroutines 7-11
NOS to NOS/VE communication 7-1
NOS/VE 7
  Task services library 3-5

O

Object file A-3
Object file list 3-5
Object libraries 3-5
Object module A-3
OF declarations C-15
OFF$DISPLAY_STATUS_ MESSAGE procedure 2-31
OFP$RECEIVE_FROM_ OPERATOR procedure 2-33
OFP$SEND_TO_OPERATOR procedure 2-32
Opening the link file 7-11
OPENLNK subroutine 7-11
Operating system name and version 2-15
Operator identifier 2-32
Operator messages 2-30
OS declarations C-15
OS name and version 2-15
OSF$PROGRAM_INTERFACE_ LIBRARY 1-3
OSF$TASK_SERVICES_ LIBRARY 3-5
OSP$APPEND_STATUS_ INTEGER procedure 6-5
OSP$APPEND_STATUS_ PARAMETER procedure 6-4
OSP$AWAIT_ACTIVITY_ COMPLETION procedure 4-2
OSP$FORMAT_MESSAGE procedure 6-12
OSP$GET_MESSAGE_LEVEL procedure 6-14
OSP$GET_STATUS_SEVERITY procedure 6-9
OSP$SET_MESSAGE_LEVEL procedure 6-15
OSP$SET_STATUS_ ABNORMAL procedure 6-3
OSP$SET_STATUS_FROM_ CONDITION procedure 6-6
Out_call_in_return condition D-6

P

P register format D-2
Page A-4
Page fault condition D-6
Page size retrieval 2-16
Parameter description format 1-12
Parameter descriptor 9-75
Parameter Descriptor Table 9-2
Parameter list 1-6
  Glossary definition A-4
  Parsing 9-1
Parameter type declarations 1-4
Parameter value A-4
Parsing
  PDT declarations 9-71
  SCL command parameter lists 9-1
  SCL function parameter lists 9-52
  SCL procedure headers 9-71
Parsing environment 9-26
Partition A-4
Partition delimiter
  Read from the link file 7-9
  Write to the link file 7-16
Path
  Description 9-21

Glossary definition A-4
Pause break handling 5-17
PDT 9-2
Declaration example 9-3
PDT Pointers 9-73
PF declarations C-23
PM declarations C-27
PMP$ABORT procedure 3-18
PMP$AWAIT_TASK_TERMINATION procedure 3-15
PMP$CAUSE_CONDITION procedure 5-22
PMP$COMPUTE_DATE_TIME procedure 2-7
PMP$CONNECT_QUEUE procedure 4-7
PMP$CONTINUE_TO_CAUSE procedure 5-15
PMP$DEFINE_QUEUE procedure 4-5
PMP$DISCONNECT_QUEUE procedure 4-8
PMP$DISESTABLISH_COND_HANDLER procedure 5-11
PMP$ENABLE_SYSTEM_CONDITIONS procedure 5-2
PMP$ESTABLISH_CONDITION_HANDLER procedure 5-9
PMP$EXECUTE procedure 3-11
PMP$EXIT procedure 3-19
PMP$FORMAT_COMPACT_DATE procedure 2-8
PMP$GENERATE_UNIQUE_NAME procedure 2-14
PMP$GET_ACCOUNT_PROJECT procedure 2-18
PMP$GET_COMPACT_DATE_TIME procedure 2-6
PMP$GET_DATE procedure 2-2
PMP$GET_JOB_MODE procedure 2-19
PMP$GET_JOB_NAMES procedure 2-20
PMP$GET_LEGIBLE_DATE_TIME procedure 2-4
PMP$GET_MICROSECOND_CLOCK procedure 2-13
PMP$GET_OS_VERSION procedure 2-15
PMP$GET_PROCESSOR_ATTRIBUTES procedure 2-16
PMP$GET_PROGRAM_DESCRIPTION procedure 3-8
PMP$GET_PROGRAM_SIZE procedure 3-7
PMP$GET_QUEUE_LIMITS procedure 4-13
PMP$GET_SRUS procedure 2-21
PMP$GET_TASK_CP_TIME procedure 2-22
PMP$GET_TASK_ID procedure 2-23
PMP$GET_TIME procedure 2-3
PMP$GET_USER_IDENTIFICATION procedure 2–24
PMP$INHIBIT_SYSTEM_CONDITIONS procedure 5-3
PMP$LOAD procedure 3-13
PMP$LOG procedure 2-29
PMP$MANAGE_SENSE_SWITCHES procedure 2-26
PMP$RECEIVE_FROM_QUEUE procedure 4-9
PMP$REMOVE_QUEUE procedure 4-6
PMP$SEND_TO_QUEUE procedure 4-11
PMP$SET_PROCESS_INTERVAL_TIMER 5-20
PMP$STATUS_QUEUE procedure 4-14
PMP$STATUS_QUEUES_DEFINED procedure 4-15
PMP$TERMINATE procedure 3-16
PMP$TEST_CONDITION_HANDLER procedure 5-23
Pointer A-4
Positioning the link file 7-9
\prefix character 9-30
Prefix character for SCL escape mode 9-30
Preprocessing procedure 9-70
Privileged instruction condition D-5

Procedure call description format 1-12
Procedure declaration example 1-7
Procedure parameter list 1-6
Procedure subcommand 9-34
Process interval timer condition handler 5-19
Process virtual address D-3
Processor attribute retrieval 2-16
Product identifiers 1-11
Program attributes 3-2
  Glossary definition A-4
Program description 3-1
  Glossary definition A-4
  Variable structure 3-6
Program execution 3-1
Program interface 7
  Library 1-3
Program library list 3-5
  Glossary definition A-4
Program services 2-1
Project name retrieval 2-18
Prompt string 9-67
PUTNLNK subroutine 7-15
PUTPLNK subroutine 7-15
Putting data in the link file 7-15
PVA D-3

Q

Qid 4-7
Queue
  Communication example 4-16
  Glossary definition A-5
  Information retrieval 4-12
  Limits 4-13
  Usage 4-4
Queue status 4-14

R

Read deadlock 7-7
Reading a command variable 8-6
Reading from the link file 7-13
Receiving
  Data from a NOS job 7-8
  Operator message 2-33
  Queue message 4-9
Record A-5
Record initialization 6-1
Reformatting a date 2-8
Reformatting a time 2-9
Related manuals 2
Removal bracket 4-4
Removing a queue 4-6
REQUEST_LINK command 7-4
Requesting the link file 7-4
Restricted command list search mode 9-30
Result library 1-4
Retrieving
  Account and project names 2-18
  Address of externally declared procedure 3-13
  Current date and time
    In compact format 2-6
    In legible format 2-4
  Current date in legible format 2-2
  Current time in legible format 2-3
  Job information 2-18
  Job mode 2-19
  Job names 2-20
  Message level 6-14
  Microsecond clock value 2-13
  Number of queues defined 4-15
  Operating system version 2-15
  Parameter list information 9-11
  Processor attributes 2-16
  Program description 3-8
  Program description size 3-7
  Queue information 4-12
  Sense switch settings 2-26
  SRUs 2-21
  System information 2-12
  Task CP time 2-22
  Task identifier 2-23
  User identification 2-24
  Working catalog 9-24
Return_option file attribute 7-5
Revision record 3
Ring A-5

Ring attribute A-5
Ring_attributes file attribute 7-5

S

Scanning
  Argument lists 9-57
  Command files 9-37
  Command lines 9-68
  Declarations 9-69
  Expressions 9-63
  Parameter lists 9-12
SCL command definition 9-1
SCL command stack 9-29
SCL interpreter 9-1
SCL procedure A-6
SCL services 8-1
Scope of a condition handler 5-4
SCU
  Directives 1-1
  Example 1-6
  Source library 1-4
Segment A-6
  Numbers 3-1
Segment access condition handler 5-89
Selectable system conditions 5-8
Sending
  Data to a NOS job 7-8
  Job status message 2-31
  Operator message 2-32
  Queue message 4-11
Sense switch management 2-25
  Example 2-27
Serial number retrieval 2-16
SET_COMMAND_LIST command 9-1
Setting an abnormal status record
  For a file identifier 6-1
  For any process identifier 6-3
  Within a condition handler 6-6
Setting the
  Message level 6-15
  Process interval timer 5-20
  Working catalog 9-25
Severity of an error 6-7
Short warning condition D-6
Simulating a condition occurrence 5-23
Soft error condition D-6
Source Code Utility 1-1
Source text preparation example 1-6
SRUs 2-21
  Glossary definition A-6
Stack frame save area D-1
Stack of command lists 9-29
Standard condition processing 5-4
Standard error message generation 6-10
Starting a NOS job 7-6
Starting a task 3-11
Starting procedure 3-5
  Glossary definition A-6
Status check 1-8
Status delimiter character 6-2
Status parameters 6-1
Status record generation 6-1
Status severity check 6-7
Status variable 1-8
Statusing a queue 4-14
Statusing queues defined 4-15
String conversion procedures 8-10
String variable initialization 8-8
Subparameter lists 9-26
Suspending a task 4-1
Switch settings 2-25
System call condition D-6
System command language services 8-1
System condition detection 5-1
System condition handler 5-13
System implementation language 5
System information retrieval 2-12
System interface
  Procedure call 1-6
  Program example 1-3
System interval timer condition D-6
System naming convention 1-11
System operator messages 2-30
System resource units 2-21
  Glossary definition A-6

System services 1-1
System standard condition processing 5-4
$SYSTEM.CYBIL.OSF$ PROGRAM_INTERFACE_ LIBRARY 1-3

T

Task
- By the task itself 3-17
- By the task that initiated the terminated task 3-14
- Communication 4-1
- Dependencies 3-14
- Execution 3-1
- Glossary definition A-6
- Initiation 3-10
- Parameters 3-9
- Stack D-1
- Status record initialization 3-19
- Suspension 4-1
- Termination 3-19

Task CP time retrieval 2-22
Task-dependent information 2-17
Task identifier retrieval 2-23
Task-independent information 2-17
Task services library 3-5
Templates 6-10
Terminate break handling 5-17
Terminating an initiated task 3-14
Testing a condition handler 5-23
Testing a parameter list for a parameter value 9-13
Testing whether a parameter value is specified as a range 9-16
Time limit handling 5-17
Token
- Definitions 9-60
- Glossary definition A-6
- Scanning 9-58

Transfer symbol 3-5
- Glossary definition A-6

Trap exception condition D-6
Type checking 1-6

U

Unimplemented instruction condition D-5
Unique name 2-14
Unlinked function 9-33
Unlinked subcommand 9-34
Unsupported file interface calls 7-9
Usage bracket 4-4
User conditions D-5
User-defined condition handler 5-21
User-defined SCL commands 9-1
User identification retrieval 2-24
User name retrieval 2-24
User_info file attribute 7-5
Using system interface procedures 1-1
- Example 1-3

Utility A-6
Utility command list search mode 9-30
Utility functions 9-52
Utility subcommands 9-36
Utility variable 8-2

V

Validation ring 8-17
Value A-7
Value count A-7
Value element A-7
Value kind specifier 9-77
Value list A-7
Value set 9-13
- Glossary definition A-8

Value set count A-8
Variable
- Initialization 8-1
- Reference 8-6
- Scope 8-2
- Value 8-7

Version number 2-15
Virtual address A-8
Virtual address space 3-1
Virtual machine identifier D-3

Virtual memory A-8
VMID D-3

W

Working catalog 9-20
  Glossary definition A-8
WREPLNK subroutine 7-16
Write deadlock 7-7
Writing a command variable 8-8
Writing a log message 2-29
Writing to the link file 7-15

X

X registers D-4
XDCL variable 8-2
XREF procedure
  declaration 1-3
XREF variable 8-2

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.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

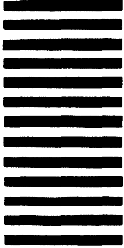
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

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



FOLD FOLD

Comments (continued from other side)

(Continued from inside front cover)

OSP$SET_MESSAGE_LEVEL 6-15
OSP$SET_STATUS_ABNORMAL 6-3
OSP$SET_STATUS_FROM_CONDITION 6-6
PMP$ABORT 3-18
PMP$AWAIT_TASK_TERMINATION 3-15
PMP$CAUSE_CONDITION 5-22
PMP$COMPUTE_DATE_TIME 2-7
PMP$CONNECT_QUEUE 4-7
PMP$CONTINUE_TO_CAUSE 5-15
PMP$DEFINE_QUEUE 4-5
PMP$DISCONNECT_QUEUE 4-8
PMP$DISESTABLISH_COND_HANDLER 5-11
PMP$ENABLE_SYSTEM_CONDITIONS 5-2
PMP$ESTABLISH_CONDITION_HANDLER 5-9
PMP$EXECUTE 3-11
PMP$EXIT 3-19
PMP$FORMAT_COMPACT_DATE 2-8
PMP$FORMAT_COMPACT_TIME 2-9
PMP$GENERATE_UNIQUE_NAME 2-14
PMP$GET_ACCOUNT_PROJECT 2-18
PMP$GET_COMPACT_DATE_TIME 2-6
PMP$GET_DATE 2-2
PMP$GET_JOB_MODE 2-19
PMP$GET_JOB_NAMES 2-20
PMP$GET_LEGIBLE_DATE_TIME 2-4
PMP$GET_MICROSECOND_CLOCK 2-13
PMP$GET_OS_VERSION 2-15
PMP$GET_PROCESSOR_ATTRIBUTES 2-16
PMP$GET_PROGRAM_DESCRIPTION 3-8
PMP$GET_PROGRAM_SIZE 3-7
PMP$GET_QUEUE_LIMITS 4-13
PMP$GET_SRUS 2-21
PMP$GET_TASK_CP_TIME 2-22
PMP$GET_TASK_ID 2-23
PMP$GET_TIME 2-3
PMP$GET_USER_IDENTIFICATION 2-24
PMP$INHIBIT_SYSTEM_CONDITIONS 5-3
PMP$LOAD 3-13
PMP$LOG 2-29
PMP$MANAGE_SENSE_SWITCHES 2-26
PMP$RECEIVE_FROM_QUEUE 4-9
PMP$REMOVE_QUEUE 4-6
PMP$SEND_TO_QUEUE 4-11
PMP$SET_PROCESS_INTERVAL_TIMER 5-20
PMP$STATUS_QUEUE 4-14
PMP$STATUS_QUEUES_DEFINED 4-15
PMP$TERMINATE 3-16
PMP$TEST_CONDITION_HANDLER 5-23