Network Operating System / Virtual Environment

(NOS/VE)

System Command Language

(SCL)

Program Interface

External Reference Specification

Revision Record

Printing	Comments
current	Made corrections to the example of a function processors.
	Added descriptions for the CLP$APPEND_STATUS_TYPE, CLP$APPEND_STATUS_VALUE_TYPE, OSP$GET_DIAGNOSTIC_SEVERITY and OSP$UNPACK_STATUS_CONDITION interfaces.
	Miscellaneous editorial corrections.
88/07/11	Minor corrections to the examples of command and function processors.
	Added descriptions for the CLP$GET_VARIABLE_VALUE, CLP$COUNT_LIST_ELEMENTS and CLP$TRIMMED_STRING_SIZE interfaces.
	Miscellaneous editorial corrections.
88/05/02	Describes the program interface to SCL at release 1.4.1 of NOS/VE. Items marked with (*) are not part of that release but may be included in a later release.
	Added a brief discussion of “work areas” which are used by a number of the program interfaces.
	Updated the CLP$INCLUDE_COMMAND request to allow more than one command to be processed.
	Corrected the description of command utility interfaces to allow for tables of new function processors.
	Added CLP$DATA_REPRESENTATION_TEXT which can be a convenience when using CLP$CONVERT_DATA_TO_STRING.
	Miscellaneous editorial corrections.
88/01/12	Changed the description of PARAMETER_DESCRIPTION_TABLE parameter of CLP$EVALUATE_PARAMETERS and CLP$EVALUATE_SUB_PARAMETERS to say that NIL may not be used to indicate that there are no allowed parameters.
	Added the VERSION parameter to the TABLE subcommand of GENERATE_COMMAND_TABLE.
	Added descriptions for interfaces CLP$APPEND_STATUS_STRING, CLP$GET_SOURCE, and CLP$GET_TYPE_INFORMATION.
	Updated the descriptions of OSP$COMPRESS_FILE_REFERENCE and OSP$EXPAND_FILE_REFERENCE.
	Miscellaneous editorial corrections.
87/10/23	Updated description of CLP$EVALUATE_PARAMETERS for new CHECK parameter attribute.
87/10/06	Added a description of the CLP$CONVERT_STRING_TO_FILE_REF request. This is the “modern” version of CLP$CONVERT_STRING_TO_FILE that instead of returning a “local file name” returns a FST$RESOLVED_FILE_REFERENCE which provides easy access to a “full path” or any of its component parts.
	Added a description of the CLP$GET_PARAMETER_NUMBER request. This interface provides for access to a command or function parameter by name. Note however, that its use introduces an overhead into the processing of parameters that is generally undesirable; in fact, it was the elimination of this overhead that, in part, lead to the new parameter processing method described in this ERS, i.e. CLP$EVALUATE_PARAMETERS.
	The EXPECTED_TYPE parameter to a CLT$FUNCTION_PROCESSOR has been removed. This information can now be obtained, in those rare cases were it is needed, by a call to the new request CLP$GET_EXPECTED_TYPE.
	Added the EVALUATION_METHOD parameter to the following requests: CLP$CREATE_ENVIRONMENT_VARIABLE (input), CLP$CREATE_PROCEDURE_VARIABLE (input), and CLP$GET_VARIABLE (output)
	Miscellaneous editorial corrections
87/09/08	Added the VARIABLE field to the CLT$PARAMETER_VALUE type, to be used for references to CLC$PASS_BY_REFERENCE (VAR) parameters.
	Changed the WORK_AREA parameter of a CLT$FUNCTION_PROCESSOR to be an “input/output” parameter rather than just “input.”

Table of Contents

1 Introduction

To assure a consistent command interface across NOS/VE and its entire range of services, a set of command language services are provided. These services support command processors, function processors, and command utilities. Before reading this section, you should have read the parts of the NOS/VE Command Interface ERS that describe the nature of commands, functions, command utilities, command language variables and the command environment.

1.1 General Considerations — Work Areas

A number of the requests that support command processing require a “work area” parameter. Most command processors need not worry about work areas since the most common request used by them, CLP$EVALUATE_PARAMETERS, provides its own. Function processors need not concern themselves with obtaining a work area since one is passed to them.

A work areas is an area of storage to be used by a request in order to do its job. This seemingly awkward interface is used because the requests involved can potentially need a lot of space to return a result (as in CLP$GET_VARIABLE or CLP$CONVERT_DATA_TO_STRING) and/or potentially need a lot of space to compute a result (as in CLP$EVALUATE_EXPRESSION or CLP$EVALUATE_SUB_PARAMETERS).

There are some cases where the size of the work area can be reasonably estimated ahead of time. Usually, however, providing a “large” work area is the safest course. Such a work area can be obtained using MMP$CREATE_SCRATCH_SEGMENT. This only needs to be done once per task. In most cases you'll want to RESET the pointer to the work area prior to passing it to an SCL request. (Function processors must not reset the work area passed to them.) The requests that use work areas, however, acknowledge the “current position” of the SEQuence pointer upon entry (i.e. the pointer is not reset before it is used) and update the “current position” to reflect how much of the work area was actually used upon exit.

2 Command Processors

2.1 clt$command

```
{
{   A command processor may receive control from the SCL interpreter either by
{ direct procedure call or by being executed as a task. Which method is used
{ is determined by how the command is made known to SCL; i.e. is the command
{ a subcommand of a utility, or a "stand-alone" program.
{
{   Regardless of how it is invoked, a command processor is passed the
{ parameters described below.
{
{       command_processor (PARAMETER_LIST, STATUS)
{
{ PARAMETER_LIST: (input)  This parameter specifies the actual parameter list
{       for the command. Command parameters are passed in a sequence for
{       compatibility with the passing of parameters to a program via the
{       pmp$execute request. Normally the command processor passes this
{       parameter on to clp$evaluate_parameters to be interpreted. If, for
{       some reason, the command processor needs to examine the unevaluated
{       parameters, it can pass this parameter to clp$get_parameter_list_text.
{
{ STATUS: (output)  This parameter specifies the completion status of the
{       command. The command processor does not need to bother with the
{       command level status parameter optionally passed to it via the
{       clt$parameter_list. The SCL interpreter will use the status passed
{       back to it in this parameter to handle the command level status
{       parameter.
{

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

*copyc cld$parameter_list
*copyc ost$status
```

2.2 Command Processor Type Declarations

2.2.1 clt$t$parameter_list

```
 TYPE
   clt$parameter_list = pmt$program_parameters;

*copyc pmt$program_parameters
```

2.2.2 clt$parameter_list_text

```
 TYPE
   clt$parameter_list_text = string ( * <= clc$max_parameter_list_size);

*copyc clc$max_parameter_list_size
```

2.2.3 clt$parameter_list_text_index

```
 TYPE
   clt$parameter_list_text_index = 1 .. clc$max_parameter_list_size + 1;

*copyc clc$max_parameter_list_size
```

2.2.4 clt$parameter_list_text_size

```
 TYPE
   clt$parameter_list_text_size = 0 .. clc$max_parameter_list_size;

*copyc clc$max_parameter_list_size

 CONST
   clc$max_parameter_list_size = clc$max_string_size;

*copyc clc$max_string_size
```

2.3 An Example

A description of the parameter processing used in the following example can be found in subsequent subsections.

```
?? NEWTITLE := 'Example Command Processor' ??
MODULE command_example;

{
{ PURPOSE:
{   The following is an example of a command processor.
{   It requires as a parameter a list of names each of which
{   is checked to see if it is a valid CYBIL identifier (SCL
{   allows more characters in names than does CYBIL.
{   Optionally, the command can be passed an integer variable,
{   name_count, which, if specified, is set to the number of
{   names in the list.
{

?? NEWTITLE := 'Global Declarations', EJECT ??
?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_list
*copyc ost$status
?? POP ??
*copyc clp$change_variable
*copyc clp$evaluate_parameters
*copyc osp$set_status_abnormal

 SECTION
   read_only: READ;

?? TITLE := 'example_command_processor', EJECT ??

 PROCEDURE [XDCL] example_command_processor
   (    parameter_list: clt$parameter_list;
    VAR status: ost$status);
```

```
{ PROCEDURE example_command (
{   names, name, n: (CHECK) list of name = $required
{   name_count, nc: (VAR) integer
{   status)

{ The text in the above comment block should be processed by
{ GENERATE_PDT (first removing the CYBIL comment delimiters)
{ and the comment block replaced by the GENERATE_PDT output.
{ In addition to the variable called "pdt", GENERATE_PDT will
{ produce the following constant declarations and the "pvt"
{ variable.

   CONST
     p$names = 1,
     p$name_count = 2,
     p$status = 3;

   VAR
     pvt: array [1 .. 3] of clt$parameter_value;


   VAR
     number_of_names: integer;

?? NEWTITLE := 'check', EJECT ??

{
{ The check procedure is called by clp$evaluate_parameters.
{ It validates that the names passed to the command are
{ legitimate CYBIL identifiers.
{

   PROCEDURE check
     (    parameter_value_table: ^clt$parameter_value_table;
          which_parameter: clt$which_parameter;
      VAR status: ost$status);

     VAR
       c: char,
       char_ok: boolean,
       cybil_id_chars: [STATIC, READ, read_only] set of
             char := ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
             'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q',
             'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
             '0', '1', '2', '3', '4', '5', '6', '7', '8',
             '9', '_', '$', '#', '@', ' '],
       cybil_id_first_chars: [STATIC, READ,
             read_only] set of char := ['A', 'B', 'C', 'D',
             'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
             'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
             'W', 'X', 'Y', 'Z', '0', '1', '2', '3', '4',
             '5', '6', '7', '8', '9'],
       i: ost$name_size,
       value: ^clt$data_value;
```

```
     IF which_parameter.specific AND
           (which_parameter.number = p$names) THEN
       number_of_names := 0;
       value := pvt [p$names].value;
       WHILE value <> NIL DO
         FOR i := 1 TO osc$max_name_size DO
           c := value^.element_value^.name_value (i);
           IF i = 1 THEN
             char_ok := c IN cybil_id_first_chars;
           ELSE
             char_ok := c IN cybil_id_chars;
           IFEND;
           IF NOT char_ok THEN

{ Since there is an unacceptable character in the name set the
{ request status to reflect the error. Then return to
{ clp$evaluate_parameters which will recall the check
{ procedure once the error has been addressed.

             osp$set_status_abnormal ('XX',
                   xxe#not_a_cybil_identifier,
                   value^.element_value^.name_value, status);
             RETURN;

           IFEND;
         FOREND;

{ Since this name is acceptable increment the name count.
{ This operation is strictly speaking not part of the
{ validation process but is done hear to avoid rescanning the
{ list in the main part of the command processor.

         number_of_names := number_of_names + 1;

         value := value^.link;
       WHILEND;
     IFEND;

   PROCEND check;
?? OLDTITLE, EJECT ??

   VAR
     value: ^clt$data_value;


{ NOTHING SHOULD BE DONE BY A COMMAND OR FUNCTION PROCESSOR
{ PRIOR TO CALLING CLP$EVALUATE_PARAMETERS.

   clp$evaluate_parameters (parameter_list, #SEQ (pdt),
         ^check, ^pvt, status);
   IF NOT status.normal THEN

{ IF CLP$EVALUATE_PARAMETERS RETURNS WITH ABNORMAL STATUS, THE
{ COMMAND OR FUNCTION PROCESSOR SHOULD IMMEDIATELY RETURN THAT
{ STATUS TO ITS CALLER.
```

```
      RETURN;
    IFEND;


    IF pvt [p$name_count].specified THEN
      PUSH value;
      value^.kind := clc$integer;

{ Number_of_names was computed in the check procedure, above.

      value^.integer_value.value := number_of_names;
      value^.integer_value.radix := 10;
      value^.integer_value.radix_specified := FALSE;
      clp$change_variable (pvt [p$name_count].variable^,
            value, status);
    IFEND;

 PROCEND example_command_processor;

MODEND command_example;
```

3 Parameter Processing

Every command and function has a Parameter Description Table (PDT) that describes the parameter names, allowed types of values, etc. This table is used to guide the evaluation of the actual parameters passed to the command or function processor.

The parameter processing techniques are designed to provide as much checking as possible for correctness at the time a command or function is called, i.e. before the processor expends too much effort attempting what has been requested of it. The responsibility for this checking is divided between the SCL interpreter and the command or function processor.

> A command or function processor should call clp$evaluate_parameters **before it does anything else**. If that interfaces returns with an abnormal status, the command or function processor should terminate, i.e. return to its caller, with that status; **and *not* do anything else**.
>
> Followng these rules will insure that the command or function fits smoothly with the rest of NOS/VE command processing including automatic parameter prompting, when selected, and calls via the DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands.

Parameter processing is accomplished by calling CLP$EVALUATE_PARAMETERS which validates that all required parameters have been supplied and checks that the values supplied for parameters meet the criteria established by the Parameter Description Table (PDT). This request takes care of the any interactions related to parameter prompting for required parameters and correction of errors. It is also responsible for preparing the information displayed by the DISPLAY_COMMAND_INFORMATION and DISPLAY_FUNCTION_INFORMATION commands.

The command processor may pass a CLT$CHECK_PARAMETERS_PROCEDURE to CLP$EVALUATE_PARAMETERS to make those checks of actual parameters that cannot be specified by the PDT. The types of checks such a procedure might make include semantic (as opposed to syntactic) validation (e.g. is the value given for a parameter meaningful in the context of the command), or validating parameter interdependancies. The CLT$CHECK_PARAMETERS_PROCEDURE is called each time a expression is successfully evaluated for a parameter by CLP$EVALUATE_PARAMETERS. It is also called once all parameters have been evaluated to allow for parameter independent checks.

Just before CLP$EVALUATE_PARAMETERS returns control to a command processor, it does appropriate logging and/or echoing for those commands that have the “manually log” log option associated with them. The image of the command call that is logged and/or echoed is edited so that secure parameters are suppressed. Calls to commands with the “automatically log” log option are logged and/or echoed prior to having their processors invoked.

The CYBIL declarations for a Parameter Description Table (PDT) can and should be produced using the GENERATE_PDT tool described in the following subsection. The PDT variable is referenced in calls to CLP$EVALUATE_PARAMETERS via CYBIL's #SEQ function. In addition to the PDT variable, GENERATE_PDT will produce CYBIL constant identifiers by which the entries in the Parameter Value Table (PVT) can be referenced. The form of the names of the constants is P$XXX where XXX is the first name given to the corresponding parameter.

3.1 generate_pdt (genpdt)

An SCL Parameter Description Table (PDT) is a sufficiently complicated data structure that its declaration and initialization in CYBIL is prohibitively awkward to attempt to do by hand. Therefore, a means of easily generating a PDT is provided.

GENERATE_PDT is an SCL tool which provides for the generation of a PDT from a specification that is identical to an SCL Procedure declaration (see the NOS/VE Command Interface ERS). This tool can also be used to generate an SCL type specification for use with the CLP$EVALUATE_EXPRESSION interface.

The output of GENERATE_PDT is a file containing the CYBIL variable declarations that represent the PDT or type specification.

```
generate_pdt, genpdt (
      input, i: file = $required
      output, o: file = $required
      page_width, pw: integer 79..110 = 79
      status)
```

INPUT, I: This parameter specifies the file containing the PDT and/or type specifications. The form of the input is described below.

The FILE_CONTENTS attribute of the INPUT file must be one of: LEGIBLE_SCL_PROCEDURE, LEGIBLE_SCL_INCLUDE, LEGIBLE_DATA or UNKNOWN.

OUTPUT, O: This parameter specifies the file which is to receive the output of this command. All lines from the input file are echoed to the output file in the form of CYBIL comments.

If GENERATE_PDT creates the OUTPUT file, its FILE_CONTENTS attribute will be set to LEGIBLE_DATA and its FILE_PROCESSOR attribute will be set to CYBIL.

PAGE_WIDTH, PW: This parameter specifies the desired page width of the output file.

Omission causes 79 to be used.

STATUS: See the section on Error Handling in the NOS/VE Command Interface ERS.

The input to GENERATE_PDT is shown below using the metalanguage notation described in section 2 of the NOS/VE Command Interface ERS.

```
<GENPDT input> ::= <bol> <pdt definition> <end of text>

<pdt definition> ::= <command proc header> <eol>
                   | <function proc header> <eol>
                   | <bol> TYPE <sp|nl>
                       <type definition> <,|;|nl>
                     TYPEND <;|nl> <eol>
```

The definitions of <command proc header>, <function proc header>, <type definition>, etc. can be found in section 2 of the NOS/VE Command Interface ERS.

The name of the variable for a PDT is always “pdt.” The name of the variable for a type specification is always “type_specification.”

For a PDT, in addition to the pdt variable declaration, a variable is declared for the parameter value table. This variable is given the name “pvt.” Also, for a PDT, a CYBIL constant is declared for each parameter. These constants can be used to index the parameter value table when referencing command or function parameters. The names of the constants are derived from the corresponding parameter names by prefixing the nominal parameter name with the characters “P$.” The nominal name for a parameter is the first of the names specified for the parameter in the PDT definition. If the resulting name is longer than 31 characters, the excess characters are truncated. If the parameter name contains any characters not allowed in a CYBIL identifier, each such character is converted to an underscore (_).

If a <procedure scope attribute> appears in a <command proc header> or a <function proc header> that is input to generate_pdt, it is ignored.

The following are examples of input to generate_pdt.

Example 1

```
PROCEDURE generate_pdt, genpdt (
 input, i: file = $input
 output, o: file = $output
 page_width, pw: integer 79..110
 status)
```

Example 2

```
TYPE
 name_and_number: record
   the_name: name
   the_number: integer 1..99
```

```
            recend
          TYPEND
```

3.2 Parameter Processing Interfaces

3.2.1 clp$evaluate_parameters

```
{
{   This request is called by a command or function processor to parse and
{ evaluate its parameter list. If an abnormal status is returned, the command
{ or function processor should return to its caller with that same abnormal
{ status. The request will verify that the actual parameters conform to the
{ specification in the Parameter Description Table (PDT).
{
{   In addition a clt$check_parameters_procedure may be given to this request.
{ If supplied, the procedure will be called to perform command or function
{ specific validation for each parameter that has the CHECK attribute and for
{ the command or function as a whole. The procedure may be called more than
{ once to perform the same validation in the context of a parameter dialog.
{
{   A command or function that has no parameters should call this request to
{ verify that no parameters were passed to it. In such a case an "empty" PDT
{ must be passed for the PARAMETER_DESCRIPTION_TABLE and NIL must be passed for
{ the CHECK_PARAMETERS_PROCEDURE and PARAMETER_VALUE_TABLE parameters.
{
{       CLP$EVALUATE_PARAMETERS (PARAMETER_LIST, PARAMETER_DESCRIPTION_TABLE,
{         CHECK_PARAMETERS_PROCEDURE, PARAMETER_VALUE_TABLE, STATUS)
{
{ PARAMETER_LIST: (input)  This parameter specifies the actual parameters
{       to be evaluated for the command or function as passed to the requesting
{       processor.
{
{ PARAMETER_DESCRIPTION_TABLE: (input)  This parameter specifies the names,
{       types and defaults for the command's or function's parameters. The
{       variable referenced for this parameter should be produced using the
{       GENERATE_PDT tool. The #SEQ function is used to pass the variable as
{       the actual value for this parametrer. NIL may not be specified to
{       indicate that there are no parameters; an "empty" PDT must be used.
{
{ CHECK_PARAMETERS_PROCEDURE: (input)  This parameter specifies the procedure
{       to be called to perform additional parameter validation. NIL may be
{       specified to indicate the absence of such a procedure.
{
{             CHECK_PARAMETERS_PROCEDURE (PARAMETER_VALUE_TABLE,
{               WHICH_PARAMETER, STATUS)
{
{       PARAMETER_VALUE_TABLE: (input)  This parameter specifies the evaluated
{             parameters for the command or function. If a parameter was
{             omitted and has no default, then the corresponding value or
{             variable field is NIL.
{
{       WHICH_PARAMETER: (input)  This parameter specifies which command or
{             function parameter to check. This parameter may specify that a
{             particular command or function parameter should be checked or
{             that general checks should be made (e.g. of parameter
{             interdependencies). Each time clp$evaluate_parameters has
{             successfully evaluated a parameter's expression (including a
```

```
{             default), it calls this routine specifying that parameter is to
{             be checked. Once all parameters have been evaluated, this
{             routine is called to perform general checks.
{
{       STATUS: (output)  This parameter specifies the result of the checking.
{
{ PARAMETER_VALUE_TABLE: (output)  This parameter specifies the evaluated
{       parameters for the command or function. NIL may be specified to
{       indicate that there are no parameters. If a parameter was omitted and
{       has no default, then the corresponding value or variable field is NIL.
{
{ STATUS: (output)  This parameter specifies the request status. If this
{       status is not normal, the command or function processor should return
{       immediately to its caller with this status.
{

 PROCEDURE [XREF] clp$evaluate_parameters
   (    parameter_list: clt$parameter_list;
        parameter_description_table: ^clt$parameter_description_table;
        check_parameters_procedure: clt$check_parameters_procedure;
        parameter_value_table: ^clt$parameter_value_table;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$check_parameters_procedure
*copyc clt$parameter_description_table
*copyc clt$parameter_list
*copyc clt$parameter_value_table
*copyc ost$status
?? POP ??
```

3.2.2 clp$get_parameter_list_text

```
{
{   The purpose of this request is to get a pointer to the text of the
{ parameter list passed to a command or function processor. The text is
{ passed in a SEQuence and this request returns a pointer to the text within
{ the sequence.
{
{   Normally this request is not needed since clp$evaluate_parameters does
{ this operation automatically. This request is provided for those processors
{ that, for whatever reason, need to directly examine the unevaluated form of
{ the parameter list passed to them.
{
{       CLP$GET_PARAMETER_LIST_TEXT (PARAMETER_LIST, PARAMETER_LIST_TEXT,
{         STATUS)
{
{ PARAMETER_LIST: (input)  This parameter specifies the parameter list
{       containing the text.
{
{ PARAMETER_LIST_TEXT: (output)  This parameter specifies the parameter list
{       text.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_parameter_list_text
   (    parameter_list: ^clt$parameter_list;
    VAR parameter_list_text: ^clt$parameter_list_text;
```

```
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_list
*copyc clt$parameter_list_text
*copyc ost$status
?? POP ??
```

3.2.3 clp$get_parameter_number

```
{
{   The purpose of this request is to determine the number (position) of a
{ command or function parameter.
{
{       CLP$GET_PARAMETER_NUMBER (PARAMETER_DESCRIPTION_TABLE, PARAMETER_NAME,
{         PARAMETER_NUMBER, STATUS)
{
{ PARAMETER_DESCRIPTION_TABLE: (input)  This parameter specifies the command's
{       or function's parameter description table (PDT).
{
{ PARAETER_NAME: (input)  This parameter specifies the name of the command or
{       function parameter whose number is desired.
{
{ PARAMETER_NUMBER: (output)  This parameter specifies the number of the
{       command or function parameter in question.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_parameter_number
   (    parameter_description_table: ^clt$parameter_description_table;
        parameter_name: clt$parameter_reference;
    VAR parameter_number: clt$parameter_number;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_description_table
*copyc clt$parameter_number
*copyc clt$parameter_reference
*copyc ost$status
?? POP ??
```

3.2.4 clp$get_reason_for_call

```
{
{   The purpose of this request is to determine why a command or function
{ processor was called. Normally, of course, the command or function is to be
{ processed; but a command or function processor may also be called as a
{ result of a call to the display_command/function_information command or its
{ equivalent. In the latter case it is the responsibility of the command or
{ function processor to display the appropriate information.
{
{   Most command and function processors need not concern themselves with this
{ aspect of interfacing with SCL since they call clp$evaluate_parameters and
{ that request takes care of it for them. This request is provided for those
{ command and function processors that, for whatever reason, don't use
{ clp$evaluate_parameters.
{
```

```
{       CLP$GET_REASON_FOR_CALL (INFORMATION_REQUEST, DISPLAY_FILE,
{         PROMPTING_ACTIVATED, STATUS)
{
{ INFORMATION_REQUEST: (output)  This parameter specifies whether the current
{       command or function processor was called to just provide information
{       (TRUE) or actually process the command or function (FALSE).
{
{ DISPLAY_FILE: (output)  This parameter specifies the file to which
{       information about the command or function is to be displayed. This
{       parameter is meaningless (and set to all blanks) when FALSE is
{       returned for INFORMATION_REQUEST.
{
{ PROMPTING_ACTIVATED: (output)  This parameter specifies whether the command
{       or function processor should prompt for missing or erroneous
{       parameters (TRUE) or not (FALSE). This parameter is meaningless (and
{       set to FALSE) when TRUE is returned for INFORMATION_REQUEST.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_reason_for_call
   (VAR information_request: boolean;
    VAR display_file: fst$path;
    VAR prompting_activated: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc fst$path
*copyc ost$status
?? POP ??
```

3.2.5 clp$log_and_or_echo_command

```
{
{   This request is intended for use by commands processors that do not use
{ clp$evaluate_parameters and have "secure" parameters. The presence of
{ "secure" parameters should cause to command to have the "manually log"
{ attribute. This attribute means that it is the responsibility of the
{ command's processor to cause the image of the command call to be logged
{ and/or echoed. A command processor that uses clp$evaluate_parameters need
{ not concern itself with this responsibility since that interface will perform
{ the operation of this request on behalf of the command processor.
{
{       CLP$LOG_AND_OR_ECHO_COMMAND (EDITED_PARAMETER_LIST_TEXT, STATUS)
{
{ EDITED_PARAMETER_LIST_TEXT: (input)  This parameter specifies the text of the
{       parameter list passed to the command processor with the "secure"
{       parameters edited out.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$log_and_or_echo_command
   (    edited_parameter_list_text: clt$parameter_list_text;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_list_text
*copyc ost$status
```

```
?? POP ??
```

3.3 “Sub-parameters”

The term “sub-parameters” is used to describe a parameter list that doesn't belong to command or function. There are two ways to deal with sub-parameters. The first is to use the CLP$EVALUATE_SUB_PARAMETERS request. This is a somewhat simplified version of the CLP$EVALUATE_PARAMETERS request and, although it has some restrictions, is usually entirely adequate for the purpose.

The second method is to use CLP$EVALUATE_PARAMETERS itself. This requires, however, that an environment be established within which that request can be called. The CLP$PUSH_PARAMETERS request is used to create this environment and the CLP$POP_PARAMETERS request to remove it once the evaluated parameters have been processed.

The CLP$EVALUATE_SUB_PARAMETERS request requires less resources and time to process, so should be chosen over the second method if its restrictions don't get in the way.

3.3.1 clp$evaluate_sub_parameters

```
{
{   This request provides for utilizing SCL's facilities to parse and evaluate
{ a parameter list other than one passed to a command or function processor.
{ It is similar to clp$evaluate_parameters except that the parameter list is
{ supplied in its text form, and no clt$check_parameters_procedure may be
{ supplied. Also the parameter description table (PDT) may not allow any VAR
{ parameters.
{
{       CLP$EVALUATE_SUB_PARAMETERS (PARAMETER_LIST_TEXT,
{         PARAMETER_DESCRIPTION_TABLE, WORK_AREA, PARAMETER_VALUE_TABLE,
{         STATUS)
{
{ PARAMETER_LIST_TEXT: (input)  This parameter specifies the parameter list
{       text to be evaluated.
{
{ PARAMETER_DESCRIPTION_TABLE: (input)  This parameter specifies the names,
{       types and defaults for the command's or function's parameters. The
{       variable referenced for this parameter should be produced using the
{       GENERATE_PDT tool. The #SEQ function is used to pass the variable as
{       the actual value for this parametrer. NIL may not be specified to
{       indicate that there are no parameters; an "empty" PDT must be used.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       will be used to construct the evaluated parameters. The current
{       position of this sequence pointer is updated to reflect the amount of
{       storage used by the request. The evaluated parameters are completely
{       contained within the used part of this sequence.
{
{ PARAMETER_VALUE_TABLE: (output)  This parameter specifies the evaluated
{       parameters. NIL may be specified to indicate that there are no
{       parameters. If a parameter was omitted and has no default, then the
{       corresponding value field is NIL.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$evaluate_sub_parameters
   (    parameter_list_text: clt$parameter_list_text;
        parameter_description_table: ^clt$parameter_description_table;
    VAR work_area {input, output} : ^clt$work_area;
```

```
        parameter_value_table: ^clt$parameter_value_table;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_description_table
*copyc clt$parameter_list_text
*copyc clt$parameter_value_table
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

3.3.2 clp$push_parameters

```
{
{   The purpose of this request is to establish an environment in which
{ clp$evaluate_parameters can be called to process a parameter list that was
{ not passed to a command or function processor. This environment is created
{ automatically for a command or function processor but must be created
{ explicitly if clp$evaluate_parameters is called for other purposes. Once
{ the "sub-parameters" have been evaluated and processed, a call should be
{ made to clp$pop_parameters to release the resources used for the parameter
{ evaluation.
{
{       CLP$PUSH_PARAMETERS (STATUS)
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] clp$push_parameters
   (VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
?? POP ??
```

3.3.3 clp$pop_parameters

```
{
{   The purpose of this request is to undo the effect of the preceding call to
{ clp$push_parameters.
{
{       CLP$POP_PARAMETERS (STATUS)
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$unexpected_call_to
{

 PROCEDURE [XREF] clp$pop_parameters
   (VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$unexpected_call_to
*copyc ost$status
?? POP ??
```

3.4 Parameter Processing Type Declarations

3.4.1 clt$application_value_text

```
 TYPE
   clt$application_value_text = clt$expression_text;

*copyc clt$expression_text
```

3.4.2 clt$boolean

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

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

3.4.3 clt$cobol_name

```
 TYPE
   clt$cobol_name = string (clc$max_cobol_name_size);

*copyc clc$max_cobol_name_size


 CONST
   clc$max_cobol_name_size = 30;
```

3.4.4 clt$check_parameters_procedure

```
 TYPE
   clt$check_parameters_procedure = ^procedure
          (    parameter_value_table: ^clt$parameter_value_table;
               which_parameter: clt$which_parameter;
           VAR status: ost$status);

*copyc clt$parameter_value_table
*copyc clt$which_parameter
*copyc ost$status
```

3.4.5 clt$cobol_name

```
 TYPE
   clt$cobol_name = string (clc$max_cobol_name_size);

*copyc clc$max_cobol_name_size
```

3.4.6 clt$command_reference

```
 TYPE
```

```
   clt$command_reference = record
     name: clt$command_name,
     case form: clt$command_reference_form of
     = clc$name_only_command_ref =
       ,
     = clc$skip_1st_entry_command_ref =
       ,
     = clc$system_command_ref =
       ,
     = clc$utility_command_ref =
       utility: clt$utility_name,
     = clc$module_or_file_command_ref =
       library_or_catalog: fst$path,
     = clc$file_cycle_command_ref =
       catalog: fst$path,
       cycle_number: fst$cycle_number,
     casend,
   recend;

*copyc clt$command_name
*copyc clt$command_reference_form
*copyc clt$utility_name
*copyc fst$cycle_number
*copyc fst$path


 TYPE
   clt$command_reference_form = (clc$name_only_command_ref,
         clc$skip_1st_entry_command_ref, clc$system_command_ref,
         clc$utility_command_ref, clc$module_or_file_command_ref,
         clc$file_cycle_command_ref);
```

3.4.7 clt$date_time

```
 TYPE
   clt$date_time = record
     value: ost$date_time,
     date_specified: boolean,
     time_specified: boolean,
   recend;

*copyc ost$date_time
```

3.4.8 clt$data_value

```
 TYPE
   clt$data_value = record
     case kind: clt$data_kind of
     = clc$application =
       application_value: ^clt$application_value_text,
     = clc$array =
       array_value: ^array [ * ] of ^clt$data_value,
     = clc$boolean =
       boolean_value: clt$boolean,
     = clc$cobol_name =
       cobol_name_value: clt$cobol_name,
     = clc$command_reference =
       command_reference_value: ^clt$command_reference,
```

```
      = clc$data_name =
        data_name_value: ost$name,
      = clc$date_time =
        date_time_value: clt$date_time,
      = clc$deferred =
        deferred_value: ^clt$expression_text,
        deferred_type: ^clt$type_specification,
      = clc$entry_point_reference =
        entry_point_reference_value: ^pmt$entry_point_reference,
      = clc$file =
        file_value: ^fst$file_reference,
      = clc$integer =
        integer_value: clt$integer,
      = clc$keyword =
        keyword_value: clt$keyword,
      = clc$list =
        element_value: ^clt$data_value,
        link: ^clt$data_value,
        generated_via_list_rest: boolean,
      = clc$lock =
        lock_value: ^clt$lock,
      = clc$name =
        name_value: ost$name,
      = clc$network_title =
        network_title_value: ^nat$title,
      = clc$program_name =
        program_name_value: pmt$program_name,
      = clc$range =
        low_value: ^clt$data_value,
        high_value: ^clt$data_value,
      = clc$real =
        real_value: clt$real,
      = clc$record =
        field_values: ^array [1 .. * ] of clt$field_value,
      = clc$scu_line_identifier =
        scu_line_identifier_value: clt$scu_line_identifier,
      = clc$statistic_code =
        statistic_code_value: sft$statistic_code,
      = clc$status =
        status_value: ^ost$status,
      = clc$status_code =
        status_code_value: ost$status_condition_code,
      = clc$string =
        string_value: ^clt$string_value,
      = clc$string_pattern =
        string_pattern_value: ^clt$string_pattern,
      = clc$time_increment =
        time_increment_value: ^pmt$time_increment,
      = clc$time_zone =
        time_zone_value: ost$time_zone,
      = clc$type_specification =
        type_specification_value: ^clt$type_specification,
      = clc$unspecified =
        ,
      casend,
    recend;

*copyc clt$application_value_text
*copyc clt$boolean
*copyc clt$cobol_name
```

```
*copyc clt$command_reference
*copyc clt$date_time
*copyc clt$data_kind
*copyc clt$expression_text
*copyc clt$field_value
*copyc clt$integer
*copyc clt$keyword
*copyc clt$lock
*copyc clt$real
*copyc clt$scu_line_identifier
*copyc clt$string_pattern
*copyc clt$string_value
*copyc clt$type_specification
*copyc fst$file_reference
*copyc nat$title
*copyc ost$name
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$time_zone
*copyc pmt$entry_point_reference
*copyc pmt$program_name
*copyc pmt$time_increment
*copyc sfd$type_declarations
```

3.4.9 clt$data_kind

```
TYPE
  clt$data_kind = (clc$application, clc$array, clc$boolean, clc$cobol_name,
        clc$command_reference, clc$data_name, clc$date_time, clc$deferred,
        clc$entry_point_reference, clc$file, clc$integer, clc$keyword,
        clc$list, clc$lock, clc$name, clc$network_title, clc$program_name,
        clc$range, clc$real, clc$record, clc$scu_line_identifier,
        clc$statistic_code, clc$status, clc$status_code, clc$string,
        clc$string_pattern, clc$time_increment, clc$time_zone,
        clc$type_specification, clc$unspecified);
```

3.4.10 clt$expression_text

```
TYPE
  clt$expression_text = string ( * <= clc$max_expression_text_size);

*copyc clc$max_expression_text_size


CONST
  clc$max_expression_text_size = clc$max_string_size;

*copyc clc$max_string_size
```

3.4.11 clt$field_name

```
TYPE
  clt$field_name = ost$name;

*copyc ost$name
```

3.4.12 clt$field_number

```
 TYPE
   clt$field_number = 1 .. clc$max_fields;

*copyc clc$max_fields
```

3.4.13 clt$field_value

```
 TYPE
   clt$field_value = record
     name: clt$field_name,
     value: ^clt$data_value,
   recend;

*copyc clt$data_value
*copyc clt$field_name
```

3.4.14 clt$integer

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

3.4.15 clt$keyword

```
 TYPE
   clt$keyword = ost$name;

*copyc ost$name
```

3.4.16 clt$lock

```
 TYPE
   clt$lock = record
     case state: clt$lock_state of
     = clc$lock_clear =
       ,
     = clc$lock_set, clc$lock_expired =
       set_by_job: jmt$system_supplied_name,
       set_by_task: pmt$task_id,
       expiration_date_time_rel_gmt: ost$date_time,
     casend,
   recend;

*copyc clt$lock_state
*copyc jmt$system_supplied_name
*copyc ost$date_time
*copyc pmt$task_id
```

3.4.17 clt$lock_state

```
TYPE
  clt$lock_state = (clc$lock_clear, clc$lock_set, clc$lock_expired);
```

3.4.18 clt$parameter_description_table

```
TYPE
  clt$parameter_description_table = SEQ ( * );

{
{ A clt$Parameter_Description_Table contains a header followed by one
{ or more of the following, depending on the header, in the order shown:
{- an array [1 .. header.number_of_parameter_names] of clt$pdt_parameter_name
{- an array [1 .. header.number_of_parameters] of clt$pdt_parameter
{- a clt$type_specification, clt$variable_name (default name), and
{ clt$expression_text (default value) for each parameter, in parameter
{ position order
{

*copyc cls$declaration_section
*copyc clt$pdt_header
*copyc clt$pdt_parameters
*copyc clt$pdt_parameter_names
```

Since the declarations for a PDT should be constructed using the GENERATE_PDT tool, the detailed types that comprise the components of a PDT are not shown here.

3.4.19 clt$parameter_passing_method

```
TYPE
  clt$parameter_passing_method = (clc$pass_by_value, clc$pass_by_reference);
```

3.4.20 clt$parameter_value

```
TYPE
  clt$parameter_value = record
    specified: boolean,
    case passing_method: clt$parameter_passing_method of
    = clc$pass_by_value =
      value: ^clt$data_value, {NIL if omitted and no default}
    = clc$pass_by_reference =
      variable: ^clt$variable_ref_expression, {NIL if omitted and no default}
    casend,
  recend;

*copyc clt$data_value
*copyc clt$parameter_passing_method
*copyc clt$variable_ref_expression
```

3.4.21 clt$parameter_value_table

```
TYPE
  clt$parameter_value_table = array [1 .. * ] of clt$parameter_value;

*copyc clt$parameter_value
```

3.4.22 clt$real

```
 TYPE
   clt$real = record
     value: longreal,
     number_of_digits: clt$real_number_digit_count,
   recend;

*copyc clt$real_number_digit_count
```

3.4.23 clt$scu_line_identifier

```
 TYPE
   clt$scu_line_identifier = record
     modification_name: clt$scu_modification_name,
     sequence_number: clt$scu_sequence_number,
   recend;

*copyc clt$scu_modification_name
*copyc clt$scu_sequence_number


 TYPE
   clt$scu_modification_name = string (clc$max_scu_modification_name);

*copyc clc$max_scu_modification_name


 CONST
   clc$max_scu_modification_name = 9;



 TYPE
   clt$scu_sequence_number = 1 .. clc$max_scu_sequence_number;

*copyc clc$max_scu_sequence_number


 CONST
   clc$max_scu_sequence_number = 0ffffff(16);
```

3.4.24 clt$string_index

```
 TYPE
   clt$string_index = 1 .. clc$max_string_size + 1;

*copyc clc$max_string_size
```

3.4.25 clt$string_pattern

```
 TYPE
   clt$string_pattern = SEQ ( * );
```

3.4.26 clt$string_size

```
 TYPE
   clt$string_size = 0 .. clc$max_string_size;

*copyc clc$max_string_size


 CONST
   clc$max_string_size = cyc$max_string_size;

*copyc cyc$max_string_size
```

3.4.27 clt$string_value

```
 TYPE
   clt$string_value = string ( * <= clc$max_string_size);

*copyc clc$max_string_size
```

3.4.28 clt$type_name

```
 TYPE
   clt$type_name = clt$variable_name;

*copyc clt$variable_name
```

3.4.29 clt$type_specification

```
 TYPE
   clt$type_specification = SEQ ( * );

 {
 { A clt$type_specification contains a clt$type_specification_header
 { followed by a qualifier for the particular type being specified.
 { For certain types the qualifier may not be present.
 {

*copyc cls$declaration_section
*copyc clt$application_type_qualifier
*copyc clt$array_type_qualifier
*copyc clt$date_time_type_qualifier
*copyc clt$integer_type_qualifier
*copyc clt$keyword_type_qualifier
*copyc clt$list_type_qualifier
*copyc clt$name_type_qualifier
*copyc clt$range_type_qualifier
*copyc clt$real_type_qualifier
*copyc clt$record_type_qualifier
*copyc clt$string_type_qualifier
*copyc clt$type_name_reference
*copyc clt$type_specification_header
*copyc clt$union_type_qualifier
```

Since the declarations for a type specification should be constructed using the GENERATE_PDT tool, the detailed types that comprise the components of a type specification are not shown here.

3.4.30 clt$which_parameter

```
 TYPE
   clt$which_parameter = record
     case specific: boolean of
     = TRUE =
       number: clt$parameter_number,
     = FALSE =
       ,
     casend,
   recend;

*copyc clt$parameter_number
```

3.4.31 fst$file_reference

```
 TYPE
   fst$file_reference = string ( * <= fsc$max_path_size);

*copyc fsc$max_path_size
```

3.4.32 ost$date_time

```
 TYPE
   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;
```

3.4.33 ost$name

```
 CONST
   osc$max_name_size = 31,
   osc$null_name = '                               ';

 TYPE
   ost$name_size = 1 .. osc$max_name_size;

 TYPE
   ost$name = string (osc$max_name_size);
```

3.4.34 pmt$entry_point_reference

```
 TYPE
   pmt$entry_point_reference = record
     entry_point: pmt$program_name,
     object_library: fst$path,
   recend;

*copyc fst$path
*copyc pmt$program_name
```

3.4.35 pmt$program_name

```
TYPE
  pmt$program_name = ost$name;

*copyc ost$name
```

3.4.36 pmt$time_increment

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

4 Command Utilities

A command utility is a command that has subcommands. The utility provides the processors for the sub-commands but calls the SCL interpreter to read the file containing calls to these commands. Thus from the utility writer's viewpoint, the SCL interpreter is part of the utility.

The benefits of this organization of a utility are substantial. The utility writer need not be concerned with the details of reading files of commands, analyzing the command images, invoking the appropriate command processor, evaluating parameters, etc. Also, the utility writer need not provide facilities for conditional or repeated execution of subcommands since the SCL control statements are available to users of the utility. By providing “built-in” functions in addition to subcommands, the utility makes available a natural way for the utility user to retrieve information which, used in conjunction with command language variables and control statements, creates what amounts to a specialized programming language.

A utility environment is defined by the CLP$BEGIN_UTILITY request and removed by a CLP$END_UTILITY request. A command utility established by CLP$BEGIN_UTILITY is considerred to belong to the task issuing that request. This means that its commands and functions can only be called from within that task.

The CLP$BEGIN_UTILITY request provides for definition of the attributes of the utility. The CLP$CHANGE_UTILITY_ATTRIBUTES request provides for changing certain of a utility's attributes. The CLP$GET_UTILITY_ATTRIBUTES request provides for retrieving certain of a utility's attributes. The attributes associated with a command utility are:

CLC$UTILITY_COMMAND_TABLE: This attribute specifies the subcommands of the utility. A utility's command table must contain at least one command—the command that terminates the utility (see CLC$UTILITY_COMMAND_TERMINATION). This table should be constructed using the GENERATE_COMMAND_TABLE tool described below.

CLC$UTILITY_SUB_CMND_LOG_ENABLED: This attribute determines whether the utility's subcommands may be logged or not. Logging of commands occurs only for commands read from he main command file (CLC$JOB_COMMAND_INPUT, i.e. $LOCAL.COMMAND) of a job. If this attribute is set to FALSE, the subcommands belonging to the utility are not logged even if they are read from that file. Commands not belonging to the utility are unaffected by this attribute. If this attribute is not specified, subcommand logging is assumed to be enabled.

CLC$UTILITY_FUNCTION_TABLE: This attribute specifies the functions supplied by the utility. This table should be constructed using the GENERATE_COMMAND_TABLE tool described below.

CLC$UTILITY_INTERACTIVE_INCLUDE: This attribute specifies the CYBIL procedure or command to be called each time an INCLUDE_FILE command or request is issued on behalf of the utility that references an interactive file. See the discussion below of interactive include processors.

CLC$UTILITY_LIBRARIES: This attribute specifies the library or libraries on which the processors for the subcommands and functions can be found. This attribute applies only to a command utility initiated by the UTILITY/UTILITYEND command and is not accepted by any of the user callable program interfaces related to command utilities. For command utilities initiated by the CLP$BEGIN_UTILITY request, the library (libraries) on which the program issuing the CLP$BEGIN_UTILITY request is (are) used to resolve calls to commands or functions whose table entries specify a call method of CLC$UNLINKED_CALLl, CLC$PROC_CALL or CLC$PROGREAM_CALL.

CLC$UTILITY_LINE_PREPROCESSOR: This attribute specifies the CYBIL procedure or command to be called each time a command line is read prior to interpreting that line. See the discussion below of line preprocessors.

CLC$UTILITY_NAME: This is a name that identifies the utility and is used to refer to the utility. It is treated as an attribute in order that the CLP$GET_UTILITY_ATTRIBUTES request can return the name of the current utility. Neither the CLP$BEGIN_UTILITY or CLP$CHANGE_UTILITY_ATTRIBUTES requests will accept this “attribute.”

CLC$UTILITY_PROMPT: This attribute specifies the prompt string to be used for interactive command input. For command lines, a “/” character is appended to the specified prompt string. For command continuation lines, the string “../” is appended to the specified prompt string. If this attribute is not specified, the utility name is used.

CLC$UTILITY_COMMAND_SEARCH_MODE: This attribute specifies the command search mode to be established when the utility is defined. Strictly speaking, the search mode is an “attribute” of the command list, not of a utility; but is provided on the CLP$BEGIN_UTILITY request because of its effect on the searching for utility supplied commands and commands from other command list entries. See the *Command List* section of the SCL Command Interface ERS for a description of these options. Upon exit from the utility, the command search mode is restored to the value it had when the utility was entered. If this attribute is not specified, CLC$GLOBAL_COMMAND_SEARCH is used.

CLC$UTILITY_COMMAND_TERMINATION: This parameter specifies the utility subcommand that is used to terminate the utility. This must be one of the commands in the command table. This attribute is not accepted by the clp$change_utility_attributes request. If this attribute is not specified, QUIT is assumed.

4.1 generate_command_table (genct)

The generate_command_table tool is used to generate tables of commands or functions for SCL command utilities. It produces CYBIL variable declarations, along with appropriate initialization, that represent command or function tables.

```
generate_command_table, generate_command_tables, genct (
      input, i: file = $required
      output, o: file = $required
      page_width, pw: integer 79..110
      status)
```

INPUT, I This parameter specifies the file from which the generator is to read the command table defintition.

The file attributes of the file must adhere to the requirements of the CLP$INCLUDE_FILE request.

OUTPUT, O: This parameter specifies the file to which the generator is to write the CYBIL declarations that represent the command or function table.

If GENERATE_COMMAND_TABLE creates this file, its FILE_CONTENTS attribute is set to LEGIBLE_DATA and its FILE_PROCESSOR attribute is set to CYBIL.

PAGE_WIDTH, PW: This parameter specifies the desired page width of the output file.

Omission causes 79 to be used.

STATUS: See the Error Handling section of the SCL Command Interface ERS.

The input to the generator is in the form of a series of “commands” that declare a new table or add entries to the “current table.”

The following comands are available: TABLE, COMMAND, FUNCTION, and TABLEND. Each TABLE “command” starts a new command or function table and is followed by any number of COMMAND “commands” or any number of FUNCTION “commands.” An empty table may be defined by omitting COMMAND and FUNCTION “commands.” A TABLEND “command” can be used to explicitly terminate the construction of a TABLE.

4.1.1 table

The TABLE “command” declares a new table.

```
table (
      name, n: name 1..24 = $required
      type, t: key
          (command, c)
          (function, f)
        keyend = command
      section_name, sn: name
      scope, s: key
          local
          xdcl
        keyend
      module, m: name
      version, v: integer 0..1 = 1
      )
```

NAME, N: This parameter specifies the name of the table. This name can be no longer than 23 characters.

TYPE, T: This parameter specifies the type of table to be generated. Either a Command (C) or Function (F) table be selected.

Omission causes a Command table to be generated.

SECTION_NAME, SN: This parameter can be used to specify a section name for the CYBIL variable declaration. The specified section must be read-only.

Omission causes no section name to be used.

SCOPE, S: This parameter specifies the scope of the CYBIL variable declaration. Specifying XDCL causes that attribute to be used in the CYBIL variable declaration. Specifying LOCAL causes the STATIC attribute to be used.

Omission causes LOCAL to be used.

MODULE, M: This parameter specifies the name to be given to a module to contain the command or function table. If this parameter is given, CYBIL module/modend statements are generated around the table declarations along with the appropriate type declarations.

Omission causes no module/modend statements or type declarations are included in the generated output.

VERSION, V: This parameter can be used to specify that an "old" style of function table be produced. Specifying one (1) selects the current function table format, and zero (0) selects the older format. This parameter is ignored for command tables.

Omission causes the current version (1) to be used.

4.1.2 tablend

The TABLEND "command" provides a way to explicitly terminate the input to generate_command_table. The end-of-information (eoi) or an end-of-partition serves the same purpose. It has no parameters.

```
tablend (
      )
```

4.1.3 command

The COMMAND "command" declares a group of entries in a command table. The order in which the names for the entries are provided has significance when the table is used by the system supplied DISPLAY_COMMAND_LIST_ENTRY command and in other similar contexts. The first entry name is taken to be the "nominal" name for the command, i.e., the name by which it is usually known. The last entry name is taken to be the "abbreviation" for the command name. Any other entry names are considered to be aliases for the command name.

```
command (
      name, names, n: list of name = $required
      processor, p: name = $required
      call_method, cm: key
          local
          xref
          load
          (procedure, proc)
          program
        keyend = local
      availability, a: key
          (normal_usage, advertised, a, nu)
          (advanced_usage, au)
          (hidden, h)
        keyend = normal_usage
      log, l: key
          (automatic, a)
          (manual, m)
        keyend = automatic
      )
```

NAME, NAMES, N: This parameter specifies the names by which the command can be called.

PROCESSOR, P: This parameter specifies the name of the processor for the command.

CALL_METHOD, CM: This parameter specifies how the processor for the command is to be invoked.

LOCAL: indicates that the processor is a CYBIL procedure in the same module as the table.

XREF: indicates that the processor is a CYBIL procedure in a separate module from the table. This option causes an XREF procedure declaration to be generated for the processor.

LOAD: indicates that the processor is a CYBIL procedure that must be dynamically loaded (e.g., via pmp$load) before it can be called.

PROCEDURE, PROC: indicates that the processor is an SCL procedure on one of the object libraries that comprise the utility.

PROGRAM: indicates that the processor is designated by a program description on one of the object libraries that comprise the utility.

Omission causes LOCAL to be used.

AVAILABILITY, A: This parameter specifies whether the command is thought to be for NORMAL_USAGE (NU) or ADVANCED_USAGE (AU), or is to be HIDDEN (H) from users. An ADVANCED_USAGE command will only appear in a display of the command list if explicitly requested. A HIDDEN command will not appear in a display of the command list.

Omission causes NORMAL_USAGE to be used.

LOG, L: This parameter determines whether the logging of a call to the command is AUTOMATIC (A), i.e. is done by the SCL interpreter, or MANUAL (M), i.e. is done by the command processor. Logging by a command processor is done in those situations where secure information potentially passed to the command should not be written to the log.

Omission causes AUTOMATIC to be used.

4.1.4 function

The FUNCTION “command” declares a group of entries in a function table. The order in which the names for the entries are provided has significance when the table is used by the system supplied DISPLAY_COMMAND_LIST_ENTRY function and in other similar contexts. The first entry name is taken to be the “nominal” name for the function, i.e., the name by which it is usually known. The last entry name is taken to be the “abbreviation” for the function name. Any other entry names are considered to be aliases for the function name.

```
function (
      name, names, n: list of name = $required
      processor, p: name = $required
      call_method, cm: key
          local
          xref
          load
          (procedure, proc)
        keyend = local
      availability, a: key
          (normal_usage, advertised, a, nu)
          (advanced_usage, au)
          (hidden, h)
        keyend = normal_usage
      )
```

NAME, NAMES, N: This parameter specifies the names by which the function can be called.

PROCESSOR, P: This parameter specifies the name of the processor for the function.

CALL_METHOD, CM: This parameter specifies how the processor for the function is to be invoked.

LOCAL: indicates that the processor is a CYBIL procedure in the same module as the table.

XREF: indicates that the processor is a CYBIL procedure in a separate module from the table. This option causes an XREF procedure declaration to be generated for the processor.

LOAD: indicates that the processor is a CYBIL procedure that must be dynamically loaded (e.g., via pmp$load) before it can be called.

PROCEDURE, PROC: indicates that the processor is an SCL procedure on one of the object libraries that comprise the program.

Omission causes LOCAL to be used.

AVAILABILITY, A: This parameter specifies whether the function is thought to be for NORMAL_USAGE (NU) or ADVANCED_USAGE (AU), or is to be HIDDEN (H) from users. An ADVANCED_USAGE function will only appear in a display of the function list if explicitly requested. A HIDDEN function will not appear in a display of the function list.

Omission causes NORMAL_USAGE to be used.

4.1.5 An example

The output of the command table generator can best be described by an example.

Example Input

```
table example_command_table..
       section_name=oss$job_paged_literal scope=xdcl
command (generate_command_table, generate_command_tables, genct)..
       processor=clp$generate_command_table..
       call_method=xref
command (display_command_list, discl)..
       processor=clp$display_command_list..
       call_method=xref

table example_function_table type=function
function $time processor=clp$$time
fucntion $peek processor=clp$$peek availability=hidden
function $file processor=clp$$file call_method=load
```

Output from Example Input

```
{ table example_command_table..
{         section_name=oss$job_paged_literal scope=xdcl
{ command (generate_command_table, generate_command_tables, genct)..
{         processor=clp$_generate_command_table..
{         call_method=xref
{ command (display_command_list, discl)..
{         processor=clp$_display_command_list..
{         call_method=xref

?? PUSH (LISTEXT := ON) ??

 VAR
   example_command_table: [XDCL, READ, oss$job_paged_literal]
         ^clt$command_table := ^example_command_table_entries,

   example_command_table_entries: [STATIC, READ,
         oss$job_paged_literal] array [1 .. 5] of
         clt$command_table_entry := [
   {} ['DISCL                          ', clc$abbreviation_entry,
         clc$normal_usage_entry, 2, clc$automatically_log,
         clc$linked_call, ^clp$_display_command_list],
   {} ['DISPLAY_COMMAND_LIST           ', clc$nominal_entry,
```

```
         clc$normal_usage_entry, 2, clc$automatically_log,
         clc$linked_call, ^clp$_display_command_list],
   {} ['GENCT                          ', clc$abbreviated_entry,
         clc$normal_usage_entry, 1, clc$automatically_log,
         clc$linked_call, ^clp$_generate_command_table],
   {} ['GENERATE_COMMAND_TABLE         ', clc$nominal_entry,
         clc$normal_usage_entry, 1, clc$automatically_log,
         clc$linked_call, ^clp$_generate_command_table],
   {} ['GENERATE_COMMAND_TABLES        ', clc$alias_entry,
         clc$normal_usage_entry, 1, clc$automatically_log,
         clc$linked_call, ^clp$_generate_command_table]];

 PROCEDURE [XREF] clp$_generate_command_table (
       parameter_list: clt$parameter_list;
   VAR status: ost$status);

 PROCEDURE [XREF] clp$_display_command_list (
       parameter_list: clt$parameter_list;
   VAR status: ost$status);

?? POP ??
{ table example_function_table type=function
{ function $time processor=clp$$time
{ fucntion $peek processor=clp$$peek availability=hidden
{ function $file processor=clp$$file call_method=load

?? PUSH (LISTEXT := ON) ??

 VAR
   example_function_table: [STATIC, READ]
         ^clt$function_processor_table := ^example_function_table_entries,
   example_function_table_entries: [STATIC, READ]
         array [1 .. 3] of
         clt$function_proc_table_entry := [
   {} ['$FILE                          ', clc$nominal_entry,
         clc$normal_usage_entry, 1, clc$linked_call, ^clp$$file],
   {} ['$PEEK                          ', clc$nominal_entry,
         clc$hidden_entry, 2, clc$linked_call, ^clp$$peek],
   {} ['$TIME                          ', clc$nominal_entry,
         clc$normal_usage_entry, 3, clc$linked_call, ^clp$$time]];

?? POP ??
```

4.2 Command Utility Interfaces

4.2.1 clp$begin_utility

```
{
{   This request defines the environment for a command utility.
{
{       CLP$BEGIN_UTILITY (NAME, ATTRIBUTES, STATUS)
{
{ NAME: (input)  This parameter specifies the name of the utility.
{
{ ATTRIBUTES: (input)  This parameter specifies the initial attributes for the
```

```
{       utility. The following attributes may be specified:
{       clc$null_utility_attribute, clc$utility_command_search_mode,
{       clc$utility_command_table, clc$utility_function_table,
{       clc$utility_function_proc_table, clc$utility_interactive_include,
{       clc$utility_line_preprocessor, clc$utility_online_manual,
{       clc$utility_prompt, clc$utility_subcmnd_log_enabled,
{       clc$utility_termination_command.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:  cle$term_command_not_defined
{                    cle$unknown_utility_attribute
{                    cle$improper_utility_attribute
{                    cle$improper_utility_attr_value
{                    cle$improper_utility_name
{

 PROCEDURE [XREF] clp$begin_utility
   (    name: clt$utility_name;
        attributes: clt$utility_attributes;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_miscellaneous
*copyc cle$ecc_utilities
*copyc clt$utility_attributes
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.2 clp$end_utility

```
{
{   This request removes the environment for a command utility.
{
{       CLP$END_UTILITY (NAME, STATUS)
{
{ NAME: (input)  This parameter specifies the name of the utiility to be
{       terminated.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:  cle$improper_utility_name
{                    cle$unexpected_call_to
{

 PROCEDURE [XREF] clp$end_utility
   (    name: clt$utility_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_miscellaneous
*copyc cle$ecc_utilities
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.3 clp$include_file

```
{
```

```
{   The purpose of this request is to interpret text contained in a specified
{ file. This request allows SCL statements contained in a specified file to
{ be processed in the same context as the issuer of the request; i.e.
{ parameter substitution, variables and condition selections are the same.
{
{   This request can be used on its own or in conjunction with the
{ clp$begin_utility request to initiate processing of the commands for a
{ utility. When used with a utility, the file to be interpreted should
{ typically be specified as clc$current_command_input which is equivalent to
{ the SCL function $COMMAND. Its use will result in the utility's commands
{ being read from the file that invoked the utility.
{
{   The text in the file may not contain incomplete statements, e.g. if a
{ WHILE statement is in the file, the corresponding WHILEND must also be in
{ the file.
{
{   The attributes of the included file must conform to the following
{ constraints.
{
{   1.  The file_contents attribute must be either LEGIBLE_DATA or UNKNOWN.
{
{   2.  The file_processor attribute must be either SCL or UNKNOWN.
{
{   3.  The record_type attribute may not be undefined (U).
{
{   4.  The access_mode attribute must contain at least the read and/or
{       execute modes.
{
{   5.  The request must be issued from a ring that is less than or equal to
{       the R2 component of the ring_attributes of the file. If the
{       access_mode attribute does not include read (i.e. the file is
{       "execute-only", the request is also constrained to be issued from a
{       ring greater than or equal to the R1 component of the ring attributes
{       of the file.
{
{   6.  A user defined file access procedure (fap) may be associated with the
{       file provided that read is in the access_mode attribute.
{
{   If the file to be included is associated with an interactive (terminal)
{ device, and the include is associated with a utility that has established an
{ "interactive include processor", that processor is called prior to
{ interpreting any commands from the file. (This call may result in the
{ utility "going into screen mode" and therein doing its own "reading" of
{ commands.)
{
{   When control is returned from the "interactive include processor", further
{ processing is determined as follows:
{
{   1.  If the returned status is abnormal, this request terminates with that
{       status.
{
{   2.  If the utility's termination command has been executed, this request
{       terminates with normal status.
{
{   3.  Otherwise, processing of commands from the interactive file takes
{       place as if there had been no "interactive include processor".
{
{
{       CLP$INCLUDE_FILE (FILE, PROMPT, UTILITY, STATUS)
{
```

```
{ FILE: (input)  This parameter specifies the file to be included.
{
{ PROMPT: (input)  This parameter specifies a prompt string to be used if the
{       file is assigned to an interactive terminal. If the request is used
{       in conjunction with a command utility, this parameter is ignored and
{       the utility's prompt attribute is used.
{
{ UTILITY: (input)  This parameter specifies the name of the utility with
{       which the clp$include_file is to be associated. Osc$null_name may be
{       specified to avoid association with any utility.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: any
{

 PROCEDURE [XREF] clp$include_file
   (    file: fst$file_reference;
        prompt: clt$prompt;
        utility: clt$utility_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clc$standard_file_names
*copyc cle$exception_condition_codes
*copyc clt$prompt
*copyc clt$utility_name
*copyc fst$file_reference
*copyc ost$status
?? POP ??
```

4.2.4 clp$include_line

```
{
{   The purpose of this request is to interpret a string as a statement list.
{ The text in the line may not contain incomplete statements, e.g. if a WHILE
{ statement is in the line, the corresponding WHILEND must also be in the line.
{
{   This request can be used on its own or in conjunction with the
{ clp$begin_utility request to initiate processing of the commands for a
{ utility.
{
{   This request differs from the clp$include_command request in that
{ clp$include_line builds a whole new "input control block" in which to process
{ commands and control statements, whereas clp$include_command processes them
{ in the context of the current "input control block".
{
{       CLP$INCLUDE_LINE (STATEMENT_LIST, UTILITY, ENABLE_ECHOING, STATUS)
{
{ STATEMENT_LIST: (input)  This parameter specifies the string to be
{       interpreted.
{
{ ENABLE_ECHOING: (input)  This parameter determines whether the interpreted
{       commands and statements may be echoed (TRUE) or not (FALSE).
{
{ UTILITY: (input)  This parameter specifies the name of the utility with which
{       the clp$include_line is to be associated. Osc$null_name may be
{       specified to avoid association with any utility.
{
{ STATUS: (output)  This parameter specifies the request status.
```

```
{

 PROCEDURE [XREF] clp$include_line
   (    statement_list: clt$command_line;
        enable_echoing: boolean;
        utility: clt$utility_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$exception_condition_codes
*copyc clt$command_line
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.5 clp$end_include

```
{
{   The purpose of this request is to terminate the interpretation of commands
{ for a command utility that was initiated by an INCLUDE_FILE command
{ (clp$include_file request) or an INCLUDE_LINE command (clp$include_line
{ request). The effect of this request is delayed until the requester returns
{ control to the command language interpreter at which time it will return to
{ the issuer of the include command or request rather than continuing to read
{ commands.
{
{       CLP$END_INCLUDE (UTILITY, STATUS)
{
{ UTILITY: (input)  This parameter specifies the utility for which command
{       interpretation is to be terminated.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$unknown_utility
{

 PROCEDURE [XREF] clp$end_include
   (    utility: clt$utility_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_utilities
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.6 clp$change_utility_attributes

```
{
{   This request is used to change the attributes of a utility that was
{ defined via the clp$begin_utility request. The attributes of a utility
{ defined via the UTILITY/UTILEND command cannot be changed via this request.
{
{       CLP$CHANGE_UTILITY_ATTRIBUTES (NAME, ATTRIBUTES, STATUS)
{
{ NAME: (input)  This parameter specifies the name of the utiility whose
{       attributes are to be changed.
{
{ ATTRIBUTES: (input)  This parameter specifies the attributes of the utility
```

```
{       that are to be changed. The following attributes may be specified:
{       clc$null_utility_attribute, clc$utility_command_table,
{       clc$utility_function_table, clc$utility_function_proc_table,
{       clc$utility_interactive_include, clc$utility_line_preprocessor,
{       clc$utility_online_manual, clc$utility_prompt,
{       clc$utility_subcmnd_log_enabled.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:  cle$unknown_utility
{                    cle$unknown_utility_attribute
{                    cle$improper_utility_attribute
{

 PROCEDURE [XREF] clp$change_utility_attributes
   (    name: clt$utility_name;
        attributes: clt$utility_attributes;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_miscellaneous
*copyc cle$ecc_utilities
*copyc clt$utility_attributes
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.7 clp$get_utility_attributes

```
{
{   This request gets the values of selected command utility attributes.
{
{       CLP$GET_UTILITY_ATTRIBUTES (NAME, ATTRIBUTES, STATUS)
{
{ NAME: (input)  This parameter specifies the name of the utiility whose
{       attributes are to be obtained. If osc$null_name is specified, the
{       currently active utility is referenced.
{
{ ATTRIBUTES: (input, output)  This parameter specifies the attributes to get
{       via setting the key field of the array elements to designate the
{       desired attribute. The attribute values are returned in the
{       corresponding array elements. The following attributes may be
{       specified:  clc$null_utility_attribute,
{       clc$utility_command_search_mode, clc$utility_command_table,
{       clc$utility_function_table, clc$utility_function_proc_table,
{       clc$utility_interactive_include, clc$utility_libraries,
{       clc$utility_line_preprocessor, clc$utility_name,
{       clc$utility_online_manual, clc$utility_prompt,
{       clc$utility_subcmnd_log_enabled, clc$utility_termination_command.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:  cle$unknown_utility
{                    cle$unknown_utility_attribute
{                    cle$improper_utility_attribute
{                    cle$improper_utility_name
{

 PROCEDURE [XREF] clp$get_utility_attributes
   (    name: clt$utility_name;
    VAR attributes {input, output} : clt$utility_attributes;
```

```
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_utilities
*copyc clt$utility_attributes
*copyc clt$utility_name
*copyc ost$status
?? POP ??
```

4.2.8 clp$get_line_from_command_file

```
{
{   The purpose of this request is to read the next "physical" line  from  the
{ current command file. Line continuation is not processed by this request.
{
{       CLP$GET_LINE_FROM_COMMAND_FILE (PROMPT_STRING, LINE, STATUS)
{
{ PROMPT_STRING: (input) This parameter specifies the string to be issued as a
{       prompt for the line if the current command file is associated with  an
{       interactive terminal.
{
{ LINE:  (output)  This  parameter  specifies  the  line. If a NIL pointer is
{       returned,  end-of-information  or  end-of-partition  on  the   current
{       command file was encounterred. If the current source of command input
{       resulted from an include_line command or request,  a  NIL  pointer  is
{       returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$get_line_from_command_file
   (    prompt_string: clt$prompt_string;
    VAR line: ^clt$command_line;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_miscellaneous
*copyc clt$command_line
*copyc clt$prompt_string
*copyc ost$status
?? POP ??
```

4.2.9 clp$collect_commands

```
{
{   The purpose of this request is to collect subsequent commands from the
{ current command file and write them, one per line, to a specified file.
{ Collection is terminated when a specified "terminator" is found as a
{ "command" (the terminator is not written to the collection file). Collection
{ is also terminated if end-of-information is encounterred.
{
{   This request is intended for use by command utilities; therefore if the
{ requesting task does not have an active call to clp$include_file/line, the
{ request will terminate with an error status.
{
{       CLP$COLLECT_COMMANDS (FILE, TERMINATOR, STATUS)
{
```

```
{ FILE: (input)  This parameter specifies the file which is to receive the
{       collected commands.
{
{ TERMINATOR: (input)  This parameter specifies the collection "terminator"
{       name.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$unexpected_call_to
{

 PROCEDURE [XREF] clp$collect_commands
   (    file: fst$file_reference;
        terminator: ost$name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc amt$local_file_name
*copyc cle$ecc_miscellaneous
*copyc fst$file_reference
*copyc ost$name
*copyc ost$status
?? POP ??
```

4.3 Command Utility Type Declarations

4.3.1 clc$standard_file_names

```
 CONST
   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$job_log                   = '$JOB_LOG                    ',
   clc$job_output                = 'OUTPUT                      ',
   clc$listing_output            = '$LIST                       ',
   clc$null_file                 = '$NULL                       ',
   clc$proc_caller_command_input = '$COMMAND_OF_CALLER          ',
   clc$standard_input            = '$INPUT                      ',
   clc$standard_output           = '$OUTPUT                     ';
```

4.3.2 clt$call_method

```
 TYPE
   clt$call_method = (clc$unspecified_call, clc$linked_call,
         clc$unlinked_call, clc$proc_call, clc$program_call);
```

4.3.3 clt$command_call_method

```
 TYPE
   clt$command_call_method = clc$linked_call .. clc$program_call;

*copyc clt$call_method
```

4.3.4 clt$command_log_option

```
TYPE
  clt$command_log_option = (clc$automatically_log, clc$manually_log);
```

4.3.5 clt$command_name

```
TYPE
  clt$command_name = ost$name;

*copyc ost$name
```

4.3.6 clt$command_search_modes

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

4.3.7 clt$command_table

```
TYPE
  clt$command_table = array [1 .. * ] of clt$command_table_entry;

*copyc clt$command_table_entry
```

Since the declarations for a command table should be constructed using the GENERATE_COMMAND_TABLE tool, the detailed types that comprise the components of a command table are not shown here.

4.3.8 clt$function_processor_table

```
TYPE
  clt$function_processor_table = array [1 .. * ] of
        clt$function_proc_table_entry;

*copyc clt$function_proc_table_entry
```

Since the declarations for a function table should be constructed using the GENERATE_COMMAND_TABLE tool, the detailed types that comprise the components of a function table are not shown here.

4.3.9 clt$prompt

```
TYPE
  clt$prompt = string ( * <= clc$max_prompt_size);

*copyc clc$max_prompt_size


TYPE
  clt$prompt_size = 0 .. clc$max_prompt_size;

*copyc clc$max_prompt_size


CONST
  clc$max_prompt_size = clc$max_prompt_string_size - 3;
```

```
*copyc clc$max_prompt_string_size
```

4.3.10 clt$prompt_string

```
 TYPE
   clt$prompt_string = string ( * <= clc$max_prompt_string_size);

*copyc clc$max_prompt_string_size


 CONST
   clc$max_prompt_string_size = ifc$max_prompt_string_size - 1;

*copyc ift$terminal_connection_types
```

4.3.11 clt$utility_attribute

```
{
{ The clt$utility_attribute data type is used to define (clp$BEGIN_utility),
{ change (clp$CHANGE_utility_attributes), and get (clp$GET_utility_attributes)
{ the attributes of a command utility. Not all of the attributes apply to
{ all of the requests. In a comment accompanying each attribute key in the
{ following type declaration are indications of which requests that attribute
{ may be used with. The indications are given by the words in upper case
{ letters in the above list of interface names.
{

 TYPE
   clt$utility_attribute = record
     case key: clt$utility_attribute_key of
     = clc$null_utility_attribute = {BEGIN, CHANGE, GET}
       ,
     = clc$utility_command_search_mode = {BEGIN, GET}
       command_search_mode: clt$command_search_modes,
     = clc$utility_command_table = {BEGIN, CHANGE, GET}
       command_table: ^clt$command_table,
     = clc$utility_function_table = {BEGIN, CHANGE, GET}
       function_table: ^clt$function_table,
     = clc$utility_function_proc_table = {BEGIN, CHANGE, GET}
       function_processor_table: ^clt$function_processor_table,
     = clc$utility_interactive_include = {BEGIN, CHANGE, GET}
       interactive_include_processor: clt$utility_interactive_in_desc,
     = clc$utility_libraries = {GET} {for use by the UTILITY/UTILEND command}
       libraries: ^array [1 .. * ] of fst$path,
     = clc$utility_line_preprocessor = {BEGIN, CHANGE, GET}
       line_preprocessor: clt$utility_line_preproc_desc,
     = clc$utility_name = {GET}
       name: clt$utility_name,
     = clc$utility_online_manual = {BEGIN, CHANGE, GET}
       online_manual_name: ost$online_manual_name,
     = clc$utility_prompt = {BEGIN, CHANGE, GET}
       prompt: clt$utility_prompt,
     = clc$utility_subcmnd_log_enabled = {BEGIN, CHANGE, GET}
       subcommand_logging_enabled: boolean,
     = clc$utility_termination_command = {BEGIN, GET}
       termination_command: clt$command_name,
     casend,
```

```
   recend;

*copyc clt$command_name
*copyc clt$command_search_modes
*copyc clt$command_table
*copyc clt$function_processor_table
*copyc clt$function_table
*copyc clt$utility_attribute_key
*copyc clt$utility_interactive_in_desc
*copyc clt$utility_line_preproc_desc
*copyc clt$utility_name
*copyc clt$utility_prompt
*copyc fst$path
*copyc ost$online_manual_name
```

4.3.12 clt$utility_attributes

```
 TYPE
   clt$utility_attributes = array [1 .. * ] of clt$utility_attribute;

*copyc clt$utility_attribute
```

4.3.13 clt$utility_attribute_key

```
 TYPE
   clt$utility_attribute_key = (clc$null_utility_attribute,
         clc$utility_command_search_mode, clc$utility_command_table,
         clc$utility_function_table, clc$utility_function_proc_table,
         clc$utility_interactive_include, clc$utility_libraries,
         clc$utility_line_preprocessor, clc$utility_name,
         clc$utility_online_manual, clc$utility_prompt,
         clc$utility_subcmnd_log_enabled, clc$utility_termination_command);
```

4.3.14 clt$utility_name

```
 TYPE
   clt$utility_name = ost$name;

*copyc ost$name
```

4.3.15 clt$utility_prompt

```
 TYPE
   clt$utility_prompt = record
     size: clt$prompt_size,
     value: string (clc$max_prompt_size),
   recend;

*copyc clc$max_prompt_size
*copyc clt$prompt_size
```

5 Function Processors

Functions are called within the context of evaluating expressions. The processing of function parameters is identical to that for command parameters (see above).

A function returns its value via the “result” parameter passed to it. A function processor must fill in all fields of the CLT$DATA_VALUE defined for the type of value being returned. See the examples below for information on how the various data values can be constructed.

5.1 clt$function_processor

```
{
{   Unlike command processors, function processors are always invoked by direct
{ CYBIL procedure call. The parameters passed to a function are the same as
{ those passed to a command processor with the addition of "work area" (which
{ is to used to construct the function's result value) and a parameter set by
{ the function processor to point to its result value.
{
{       function_processor (PARAMETER_LIST, WORK_AREA, RESULT, STATUS)
{
{ PARAMETER_LIST: (input)  This parameter specifies the actual parameter list
{       for the function. Function parameters are passed in a sequence for
{       compatibility with the passing of parameters to a command processor.
{       Normally the function processor passes this parameter on to
{       clp$evaluate_parameters to be interpreted. If, for some reason, the
{       function processor needs to examine the the unevaluated parameters, it
{       can pass this parameter to clp$get_parameter_list_text.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       must be used to construct the function's result value. The current
{       position of this sequence pointer must be updated to reflect the amount
{       of storage used by the function. The function's result value must be
{       completely contained within the used part of this sequence.
{
{ RESULT: (output)  This parameter specifies the function's result value.
{
{ STATUS: (output)  This parameter specifies the completion status of the
{       function.
{

 TYPE
   clt$function_processor = ^procedure
           (    parameter_list: clt$parameter_list;
            VAR work_area {input, output} : ^clt$work_area;
            VAR result: ^clt$data_value;
            VAR status: ost$status);

*copyc clt$data_value
*copyc clt$parameter_list
*copyc clt$work_area
*copyc ost$status
```

5.2 An Example

```
?? NEWTITLE := 'Example SCL function processor.' ??
```

```
MODULE function_example;

{ PURPOSE:
{   This example of a function processor illustrates the
{   construction of the various types of values. The function
{   parameter is the name of the generic type of data to be
{   returned.

?? NEWTITLE := 'Global Declarations', EJECT ??
?? PUSH (LISTEXT := ON) ??
*copyc clt$data_value
*copyc clt$parameter_list
*copyc clt$work_area
*copyc ost$status
?? POP ??
*copyc clp$evaluate_parameters
*copyc pmp$get_compact_date_time

 SECTION
   read_only: READ;

?? TITLE := 'generic_data_function', EJECT ??

 PROCEDURE [XDCL] generic_data_function
   (    parameter_list: clt$parameter_list;
    VAR work_area {input, output} : ^clt$work_area;
    VAR result: ^clt$data_value;
    VAR status: ost$status);

{ FUNCTION $generic_data (
{       data_type: name = $required
{       )

{ The above comment block should be run through GENERATE_PDT
{ (with the comment delimiters removed and the output used to
{ replace those comments. In addition to the variable called
{ "pdt", GENERATE_PDT will produce the following constant
{ declaration and the "pvt" variable.

   CONST
     p$data_type = 1;

   VAR
     pvt: array [1 .. 1] of clt$parameter_value;


   VAR
     name_table: [READ, read_only] array [1 .. 3] of
           ost$name := ['LUDWIG', 'AYN', 'SERGEI'];

   VAR
     data_type: ost$name,
     i: integer,
     value: ^clt$data_value;


{ NOTHING SHOULD BE DONE BY A COMMAND OR FUNCTION PROCESSOR
{ PRIOR TO CALLING CLP$EVALUATE_PARAMETERS.

   clp$evaluate_parameters (parameter_list, #SEQ (pdt), NIL,
```

```
         ^pvt, status);
   IF NOT status.normal THEN

{ IF CLP$EVALUATE_PARAMETERS RETURNS WITH ABNORMAL STATUS, THE
{ COMMAND OR FUNCTION PROCESSOR SHOULD IMMEDIATELY RETURN THAT
{ STATUS TO ITS CALLER.

     RETURN;
   IFEND;

   data_type := pvt [p$data_type].value^.name_value;

   NEXT result IN work_area;

   IF data_type = 'APPLICATION' THEN
     result^.kind := clc$application;
     NEXT result^.application_value: [33] IN work_area;
     result^.application_value^ :=
           '>-Some( gobbledy gook ):or-other!';

   ELSEIF data_type = 'ARRAY' THEN
     result^.kind := clc$array;

{ Construct an array of three integers.

     NEXT result^.array_value: [-1 .. 1] IN work_area;
     FOR i := LOWERBOUND (result^.array_value^)
           TO UPPERBOUND (result^.array_value^) DO
       NEXT result^.array_value^ [i] IN work_area;
       result^.array_value^ [i]^.kind := clc$integer;
       result^.array_value^ [i]^.integer_value.value := i;
       result^.array_value^ [i]^.integer_value.radix := 10;
       result^.array_value^ [i]^.integer_value.
             radix_specified := FALSE;
     FOREND;

   ELSEIF data_type = 'BOOLEAN' THEN
     result^.kind := clc$boolean;
     result^.boolean_value.value := TRUE;

{ The kind field must be set and is used if the value is
{ converted to its string representation (e.g. displayed).

     result^.boolean_value.kind := clc$yes_no_boolean;

   ELSEIF data_type = 'COBOL_NAME' THEN
     result^.kind := clc$cobol_name;
     result^.cobol_name_value := 'A-COBOL-NAME';

   ELSEIF data_type = 'COMMAND_REFERENCE' THEN
     NEXT result^.command_reference_value IN work_area;
     result^.kind := clc$command_reference;
     result^.command_reference_value^.name :=
           'DISPLAY_VALUE';
     result^.command_reference_value^.form :=
           clc$system_command_ref;

   ELSEIF data_type = 'DATA_NAME' THEN
     result^.kind := clc$data_name;
     result^.data_name_value := '$GENERIC_DATA';
```

```
   ELSEIF data_type = 'DATE' THEN
     result^.kind := clc$date_time;
     pmp$get_compact_date_time (result^.date_time_value.
           value, status);
     result^.date_time_value.date_specified := TRUE;
     result^.date_time_value.time_specified := FALSE;
     result^.date_time_value.value.hour := 0;
     result^.date_time_value.value.minute := 0;
     result^.date_time_value.value.second := 0;
     result^.date_time_value.value.millisecond := 0;

   ELSEIF data_type = 'DATE_TIME' THEN
     result^.kind := clc$date_time;
     pmp$get_compact_date_time (result^.date_time_value.
           value, status);
     result^.date_time_value.date_specified := TRUE;
     result^.date_time_value.time_specified := TRUE;

   ELSEIF data_type = 'ENTRY_POINT_REFERENCE' THEN
     result^.kind := clc$entry_point_reference;
     NEXT result^.entry_point_reference_value IN work_area;
     result^.entry_point_reference_value^.entry_point :=
           'START_PROGRAM';
     result^.entry_point_reference_value^.object_library :=
           '$USER.MY_LIB';

   ELSEIF data_type = 'FILE' THEN
     result^.kind := clc$file;
     NEXT result^.file_value: [26] IN work_area;
     result^.file_value^ := ':MY_FAMILY.MY_NAME.MY_FILE';

   ELSEIF data_type = 'INTEGER' THEN
     result^.kind := clc$integer;
     result^.integer_value.value := 0ffff(16);

{ The radix and radix_specified fields must be set and are
{ used if the value is converted to its string representation
{ (e.g. displayed).

     result^.integer_value.radix := 16;
     result^.integer_value.radix_specified := TRUE;

   ELSEIF data_type = 'KEYWORD' THEN
     result^.kind := clc$keyword;
     result^.keyword_value := 'A_KEYWORD';

   ELSEIF data_type = 'LINE_IDENTIFIER' THEN
     result^.kind := clc$scu_line_identifier;
     result^.scu_line_identifier_value.modification_name :=
           'ORIGINAL';
     result^.scu_line_identifier_value.sequence_number :=
           123;

   ELSEIF data_type = 'LIST' THEN
     value := result;

{ Construct a list from an array of names.

     FOR i := LOWERBOUND (name_table)
```

```
          TO UPPERBOUND (name_table) DO
        value^.kind := clc$list;
        NEXT value^.element_value IN work_area;
        value^.element_value^.kind := clc$name;
        value^.element_value^.name_value := name_table [i];
        value^.generated_via_list_rest := FALSE;
        IF i < UPPERBOUND (name_table) THEN
          NEXT value^.link IN work_area;
          value := value^.link;
        ELSE
          value^.link := NIL;
        IFEND;
      FOREND;

   ELSEIF data_type = 'LOCK' THEN
     result^.kind := clc$lock;
     NEXT result^.lock_value IN work_area;
     result^.lock_value^.state := clc$lock_clear;

   ELSEIF data_type = 'NAME' THEN
     result^.kind := clc$name;
     result^.name_value := 'A_NAME';

   ELSEIF data_type = 'NETWORK_TITLE' THEN
     result^.kind := clc$network_title;
     NEXT result^.network_title_value: [10] IN work_area;
     result^.network_title_value^ := 'VE_860_109';

   ELSEIF data_type = 'PROGRAM_NAME' THEN
     result^.kind := clc$program_name;
     result^.program_name_value := 'A-program/"Name"';

   ELSEIF data_type = 'RANGE' THEN
     result^.kind := clc$range;

{ Construct a range of integer values.

     NEXT result^.low_value IN work_area;
     result^.low_value^.kind := clc$integer;
     result^.low_value^.integer_value.value := -123;
     result^.low_value^.integer_value.radix := 10;
     result^.low_value^.integer_value.radix_specified :=
           FALSE;
     NEXT result^.high_value IN work_area;
     result^.high_value^.kind := clc$integer;
     result^.high_value^.integer_value.value := 123;
     result^.high_value^.integer_value.radix := 10;
     result^.high_value^.integer_value.radix_specified :=
           FALSE;

   ELSEIF data_type = 'REAL' THEN
     result^.kind := clc$real;
     result^.real_value.value := 123.456d0;

{ The number_of_digits field  must be set and is used if the
{ value is converted to its string representation (e.g.
{ displayed). If the number of digits isn't known, use
{ clc$max_real_number_digits.

     result^.real_value.number_of_digits := 6;
```

```
{ See the ELSE clause below for a RECORD type example.

   ELSEIF data_type = 'STATISTIC_CODE' THEN
     result^.kind := clc$statistic_code;
     result^.statistic_code_value := xxl#some_statistic;

   ELSEIF data_type = 'STATUS' THEN
     result^.kind := clc$status;
     NEXT result^.status_value IN work_area;
     result^.status_value^.normal := TRUE;

   ELSEIF data_type = 'STATUS_CODE' THEN
     result^.kind := clc$status_code;
     result^.status_code_value := xxe#some_status_condition;

   ELSEIF data_type = 'STRING' THEN
     result^.kind := clc$string;
     NEXT result^.string_value: [8] IN work_area;
     result^.string_value^ := 'a string';

   ELSEIF data_type = 'TIME' THEN
     result^.kind := clc$date_time;
     pmp$get_compact_date_time (result^.date_time_value.
           value, status);
     result^.date_time_value.date_specified := FALSE;
     result^.date_time_value.time_specified := TRUE;

   ELSEIF data_type = 'TIME_INCREMENT' THEN
     result^.kind := clc$time_increment;
     NEXT result^.time_increment_value IN work_area;
     result^.time_increment_value^.year := 0;
     result^.time_increment_value^.month := 0;
     result^.time_increment_value^.day := 0;
     result^.time_increment_value^.hour := 0;
     result^.time_increment_value^.minute := 30;
     result^.time_increment_value^.second := 0;
     result^.time_increment_value^.millisecond := 0;

   ELSEIF data_type = 'TIME_ZONE' THEN
     result^.kind := clc$time_zone;
     result^.time_zone_value.hours_from_gmt := -6;
     result^.time_zone_value.minutes_offset := 0;
     result^.time_zone_value.daylight_saving_time := TRUE;

   ELSE { assume RECORD type }
     result^.kind := clc$record;

{ Construct a record value whose type specification is
{     RECORD
{       name_field: name
{       number_field: integer
{     RECEND

     NEXT result^.field_values: [1 .. 2] IN work_area;
     result^.field_values^ [1].name := 'NAME_FIELD';
     NEXT result^.field_values^ [1].value IN work_area;
     result^.field_values^ [1].value^.kind := clc$name;
     result^.field_values^ [1].value^.name_value := 'A_NAME';
     result^.field_values^ [2].name := 'NUMBER_FIELD';
```

```
     NEXT result^.field_values^ [2].value IN work_area;
     result^.field_values^ [2].value^.kind := clc$integer;
     result^.field_values^ [2].value^.integer_value.value :=
           123;
     result^.field_values^ [2].value^.integer_value.radix :=
           10;
     result^.field_values^ [2].value^.integer_value.
           radix_specified := FALSE;

   IFEND;

 PROCEND generic_data_function;

MODEND function_example;
```

5.3 clp$get_expected_type

```
{
{   The purpose of this request is to determine the type of result that a
{ function is expected to return. This request is intended for functions like
{ $READ which base the form of their results on the expected type.
{
{       CLP$GET_EXPECTED_TYPE (WORK_AREA, EXPECTED_TYPE, STATUS)
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       is used to construct the expected type's specification. The position
{       of this sequence pointer is be updated to reflect the amount of storage
{       used by the request. Normally, the work_area parameter passed to the
{       function processor is used for this parameter.
{
{ EXPECTED_TYPE: (output)  This parameter specifies the specification of the
{       expected type of the function's result. If NIL is returned, no
{       particular tpye of result is expected, i.e. any type of result is
{       be acceptable.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_expected_type
   (VAR work_area {input, output} : ^clt$work_area;
    VAR expected_type: ^clt$type_specification;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$type_specification
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

6 Application Values

It is sometimes necessary for an application (command utility or individual command) to have a kind of parameter value not directly supported in SCL. To accomodate this need, SCL provides a “hook” known as an “application value.” The responsibility for evaluating an expression for an application value is that of the command or function processor that accepts such a value. The syntax of an application value is constrained as described in section 2 of the NOS/VE Command Interface ERS in the subsection on Application Types. Apart from those constraints, the SCL interpreter assumes no knowledge of the syntax or semantics of an application value.

A CLC$APPLICATZION_VALUE is represented in a CLT$DATA_VALUE as a string.

7 Interactive Include Processors

An interactive include processor may be implemented either as a command, as described in the SCL Command Interface ERS (e.g. as an SCL procedure), or as a CYBIL procedure, as described below.

7.1 clt$utility_interactive_include

```
{
{   A command utility may choose to be given control each time an include file
{ command or request that references an interactive (terminal) file is issued
{ on behalf of the utility. This gives the utility the opportunity to switch
{ to "screen mode" processing automatically at the appropriate point in time.
{
{   When control is returned from the "interactive include processor", further
{ processing is determined as follows:
{
{   1.  If the returned status is abnormal, the include file command or request
{       terminates with that status.
{
{   2.  If the utility's termination command has been executed, the include
{       file command or request terminates with normal status.
{
{   3.  Otherwise, processing of commands from the interactive file takes place
{       as if there had been no "interactive include processor".
{
{
{       interactive_line_processor (INTERACTION_STYLE, STATUS)
{
{ INTERACTION_STYLE: (input)  This parameter specfies the user selected
{       interaction style at the point at which the include file command or
{       request was issued.
{
{ STATUS: (output)  This parameter specifies the interactive include
{       processor's termination status.
{

 TYPE
   clt$utility_interactive_include = ^procedure
           (    interaction_style: ost$interaction_style;
            VAR status: ost$status);

*copyc ost$interaction_style
*copyc ost$status
```

7.2 clt$utility_interactive_in_desc

```
 TYPE
   clt$utility_interactive_in_desc = record
     case call_method: clt$call_method of
     = clc$unspecified_call =
       {this option specifies the absence of an interactive include processor}
       ,
     = clc$linked_call =
       proc: clt$utility_interactive_include,
     = clc$unlinked_call =
```

```
        procedure_name: pmt$program_name,
      = clc$proc_call, clc$program_call =
        command_name: pmt$program_name,
      casend,
    recend;

*copyc clt$call_method
*copyc clt$utility_interactive_include
*copyc pmt$program_name
```

8 Command Line Preprocessors

A line preprocessor may be implemented either as a command, as described in the SCL Command Interface ERS (e.g. as an SCL procedure), or as a CYBIL procedure, as described below.

8.1 clt$utility_line_preprocessor

```
{
{   A command utility may choose to be given control each time a command line
{ is read. This "hook" is provided for those utilities whose input is not
{ completely compatible with SCL. Such incompatibility may result from a need
{ to meet an industry standard or retain compatibility with a predecessor
{ product. Also, a utility may simply wish to be notified that a command line
{ has been read; for example, a line boundary may be a convenient point to
{ which to back up for a utility that supports an "undo" feature.
{
{   This capability allows a utility to use SCL facilities for reading
{ commands when that would otherwise not be possible. Also, users of such a
{ utility may still be able to make use of SCL's programming language
{ facilities in conjunction with the utility.
{
{       utility_line_preprocessor (COMMAND_LINE, INTERACTIVE_SOURCE,
{         INTERPRETING_COMMANDS, EDITED_COMMAND_LINE, STATUS)
{
{ COMMAND_LINE: (input)  This parameter specifies the command line read by the
{       SCL interpreter.
{
{ INTERACTIVE_SOURCE: (input)  This parameter specfies whether the command
{       line originated from an interactive terminal (TRUE) or not (FALSE).
{
{ INTERPRETING_COMMANDS: (input)  This parameter specifies whether the SCL
{       interpreter would interpret (TRUE) the next command or skip it
{       (FALSE). Skip mode is entered as a result of various control
{       statements, for example IF, ELSE, etc. If the utility processes the
{       commands in the command line itself, it should only do so when this
{       parameter is TRUE.
{
{ EDITED_COMMAND_LINE: (output)  This parameter specifies the string that
{       represents the preprocessed command line. (Upon entry to the line
{       preprocessor, the edited_command_line variable is NIL.) The
{       preprocessor may construct a new line in any manner it chooses and set
{       this parameter to point to it. If this parameter is NIL upon return
{       to the SCL interpreter, the original command line is processed. If
{       this parameter points to a null (empty) string, the SCL interpreter
{       will do no further processing with this line.
{
{ STATUS:   (output)   This   parameter   specifies  the  line  preprocessor's
{       termination status.
{

 TYPE
   clt$utility_line_preprocessor = ^procedure
           (    command_line: ^clt$command_line;
                interactive_source: boolean;
                interpreting_commands: boolean;
            VAR edited_command_line: ^clt$command_line;
```

```
          VAR status: ost$status);

*copyc clt$command_line
*copyc ost$status
```

8.2 clt$utility_line_preproc_desc

```
 TYPE
   clt$utility_line_preproc_desc = record
     case call_method: clt$call_method of
     = clc$unspecified_call =
       , {this option specifies the absence of a line preprocessor}
     = clc$linked_call =
       proc: clt$utility_line_preprocessor,
     = clc$unlinked_call =
       procedure_name: pmt$program_name,
     = clc$proc_call, clc$program_call =
       command_name: pmt$program_name,
     casend,
   recend;

*copyc clt$call_method
*copyc clt$utility_line_preprocessor
*copyc pmt$program_name
```

9 Command Language Variables and Environment Objects

Command language variables provide the facility to save values of various kinds in named containers. A complete description of the command language variable can be found in the SCL Command Interface ERS.

In addition to variables the environment established by the SCL interpreter consists of a number of items referred to as environment objects. Each such object has a number of commands, functions and/or requests provided for its manipulation.

9.1 Variable and Environment Interfaces

9.1.1 clp$create_environment_variable

```
{
{   This request creates an SCL "environment" variable.
{
{       CLP$CREATE_ENVIRONMENT_VARIABLE (NAME, SCOPE, ACCESS_MODE,
{         EVALUATION_METHOD, TYPE_SPECIFICATION, INITIAL_VALUE, STATUS)
{
{ NAME: (input)  This parameter specifies the name of the variable to be
{       created.
{
{ SCOPE: (input)  This parameter specifies the scope (residence) of the
{       environment variable. The options are:
{
{       clc$environment_scope:  the variable is to be created in the current
{             block
{
{       clc$utility_scope:  the variable is to be created in the most recently
{             activated utility block
{
{       clc$task_scope:  the variable is to be created in the most recently
{             activated task block
{
{       clc$job_scope:  the variable is to be created in the job block
{
{       clc$push_scope:  this is, strictly speaking, not a true scope. It is
{             provided as a convenience for defining environment variables in
{             a context where a definition may or may not have already been
{             made.
{
{             If the variable has already been defined as an environment
{             variable, it is PUSHed (see the clp$push_environment request for
{             information on pushing an environemnt variable). The type must
{             be identical to the original definition. If an initial value is
{             specified, it becomes the value of the PUSHed instance of the
{             variable.
{
{             If the variable has not already been defined, it is defined with
{             a scope of clc$environment_scope.
{
{ ACCESS_MODE: (input)  This parameter specifies how the variable, once
{       created, may be accessed.
{
{ EVALUATION_METHOD: (input)  This parameter specifies whether values assigned
{       to the variable are to be evaluated at the time of the assignment
{       (clc$immediate_evaluation) or at the time the variable is referenced
```

```
{       (clc$deferred_evaluation). Immediate evaluation is the normal choice.
{
{ TYPE_SPECIFICATION: (input)  This parameter specifies the type of the
{       variable to be created.
{
{ INITIAL_VALUE: (input)  This parameter specifies the initial value to to be
{       given to the created variable. If clc$read_write is specified for
{       ACCESS_MODE, no initial value need be supplied (indicated by giving a
{       NIL pointer). If clc$read_only is specified, an initial value for the
{       variable must be supplied.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$create_environment_variable
   (    name: clt$variable_name_reference;
        scope: clt$environment_variable_scope;
        access_mode: clt$data_access_mode;
        evaluation_method: clt$expression_eval_method;
        type_specification: ^clt$type_specification;
        initial_value: ^clt$data_value;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$var_already_created
*copyc clt$data_access_mode
*copyc clt$data_value
*copyc clt$environment_variable_scope
*copyc clt$expression_eval_method
*copyc clt$type_specification
*copyc clt$variable_name_reference
*copyc ost$status
?? POP ??
```

9.1.2 clp$create_procedure_variable

```
{
{   This request creates an SCL "procedure" variable.
{
{       CLP$CREATE_PROCEDURE_VARIABLE (NAME, SCOPE, ACCESS_MODE,
{         EVALUATION_METHOD, TYPE_SPECIFICATION, INITIAL_VALUE, STATUS)
{
{ NAME: (input) This parameter specifies  the  name  of  the  variable  to  be
{       created.
{
{ SCOPE:  (input)  This  parameter  specifies the scope (accessibility) of the
{       procedure variable. The options are:
{
{       clc$local_scope: the variable is to  be  accessible  only  within  the
{             current block
{
{       clc$xdcl_scope:  the  variable  is  to  be  accessible  by  any  block
{             performing a corresponding XREF declaration.
{
{       clc$xref_scope: the variable to be created is actually a reference  to
{             a variable created with a corresponding XDCL declaration.
{
{ ACCESS_MODE:  (input)  This  parameter  specifies  how  the  variable,  once
{       created, may be accessed.
```

```
{
{ EVALUATION_METHOD: (input)  This parameter specifies whether values assigned
{       to the variable are to be evaluated at the time of the assignment
{       (clc$immediate_evaluation) or at the time the variable is referenced
{       (clc$deferred_evaluation). Immediate evaluation is the normal choice.
{
{ TYPE_SPECIFICATION:  (input)  This  parameter  specifies  the  type  of  the
{       variable to be created.
{
{ INITIAL_VALUE:  (input)  This specifies the initial value to be given to the
{       created variable. If clc$read_write is specified for ACCESS_MODE,  no
{       initial  value  need  be supplied (indicated by giving a NIL pointer).
{       If clc$read_only is specified, an initial value for the variable  must
{       be  supplied.  If  clc$xref_scope  is specified for SCOPE, no initial
{       value may be supplied.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$create_procedure_variable
   (    name: clt$variable_name_reference;
        scope: clt$procedure_variable_scope;
        access_mode: clt$data_access_mode;
        evaluation_method: clt$expression_eval_method;
        type_specification: ^clt$type_specification;
        initial_value: ^clt$data_value;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$var_already_created
*copyc clt$data_access_mode
*copyc clt$data_value
*copyc clt$expression_eval_method
*copyc clt$procedure_variable_scope
*copyc clt$type_specification
*copyc clt$variable_name_reference
*copyc ost$status
?? POP ??
```

9.1.3 clp$delete_variable

```
{
{   The purpose  of this request is to delete a command language variable from
{ the current block.
{
{       CLP$DELETE_VARIABLE (NAME, STATUS)
{
{ NAME: (input) This parameter specifies  the  name  of  the  variable  to  be
{       deleted.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$improper_variable_name, cle$unknown_variable.
{       IDENTIFIER: 'CL'
{

 PROCEDURE [XREF] clp$delete_variable
   (    name: string ( * );
    VAR status: ost$status);
```

```
?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_variable
*copyc ost$status
?? POP ??
```

9.1.4 clp$change_variable

```
{
{   This request changes the value of an SCL variable. Any variable reference
{ permitted  as  the object of an SCL assignment statement may be specified as
{ the variable to be changed.
{
{       CLP$CHANGE_VARIABLE (REFERENCE, VALUE, STATUS)
{
{ REFERENCE: (input) This parameter specifies the variable to be changed.
{
{ VALUE: (input) This parameter specifies the new value for the variable.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$change_variable
   (    reference: clt$variable_ref_expression;
        value: ^clt$data_value;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$unknown_variable
*copyc clt$data_value
*copyc clt$variable_ref_expression
*copyc ost$status
?? POP ??
```

9.1.5 clp$get_variable

```
{
{   This request retrieves the value, and related information, for an SCL
{ variable or variable component. Any variable reference may be used to
{ specify the variable to be interrogated.
{
{       CLP$GET_VARIABLE (REFERENCE, WORK_AREA, CLASS, ACCESS_MODE,
{         EVALUATION_METHOD, TYPE_SPECIFICATION, VALUE, STATUS)
{
{ REFERENCE: (input)  This parameter specifies the variable to be interrogated.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage into
{       which is copied the returned variable's type specification and value.
{       The current position of this sequence pointer is updated to reflect
{       the amount of storage used by the request.
{
{ CLASS: (output)  This parameter specifies the class of the variable.
{
{ ACCESS_MODE: (output)  This parameter specifies the access mode of the
{       variable.
{
{ EVALUATION_METHOD: (output)  This parameter specifies the evaluation method
{       for the variable.
```

```
{
{ TYPE_SPECIFICATION: (output)  This parameter specifies (a copy of) the type
{       specification for the variable or variable component.
{
{ VALUE: (output)  This parameter specifies (a copy of) the value of the
{       variable or variable component. If the variable has never been
{       assigned a value and the reference is to the entire variable (as
{       opposed to a component of the variable), NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_variable
   (    reference: clt$variable_ref_expression;
    VAR work_area {input, output} : ^clt$work_area;
    VAR class: clt$variable_class;
    VAR access_mode: clt$data_access_mode;
    VAR evaluation_method: clt$expression_eval_method;
    VAR type_specification: ^clt$type_specification;
    VAR value: ^clt$data_value;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$unknown_variable
*copyc clt$data_access_mode
*copyc clt$data_value
*copyc clt$expression_eval_method
*copyc clt$type_specification
*copyc clt$variable_class
*copyc clt$variable_ref_expression
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

9.1.6 clp$get_variable_value

```
{
{   This request retrieves the value for an SCL variable or variable component.
{ Any variable reference may be used to specify the variable.
{
{   The VALUE parameter returned by this requests points to a copy of the
{ variable's value. The space occupied by that copy is released when the
{ requesting command processor terminates. Therefore if a variable's value is
{ to be retreived repeatedly during the execution of a command processor it may
{ be better to use the CLP$GET_VARIABLE request which provides a WORK_AREA
{ parameter, thereby allowing the space to be reused on each call.
{
{       CLP$GET_VARIABLE_VALUE (REFERENCE, VALUE, STATUS)
{
{ REFERENCE: (input)  This parameter specifies the variable to be interrogated.
{
{ VALUE: (output)  This parameter specifies (a copy of) the value of the
{       variable or variable component. If the variable has never been
{       assigned a value and the reference is to the entire variable (as
{       opposed to a component of the variable), NIL is returned.
{
{ STATUS: (output)  This parameter specifies the request status.
{
```

```
 PROCEDURE [XREF] clp$get_variable_value
   (    reference: clt$variable_ref_expression;
    VAR value: ^clt$data_value;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$unknown_variable
*copyc clt$data_value
*copyc clt$variable_ref_expression
*copyc ost$status
?? POP ??
```

9.1.7 clp$push_environment

```
{
{   This request pushes the current instance of an  SCL  "environment  object"
{ and  creates  a  new  instance of that object. The initial value of the new
{ instance of the object is the current value of the pushed instance.
{
{       CLP$PUSH_ENVIRONMENT (OBJECT, STATUS)
{
{ OBJECT: (input) This parameter specifies the name of the environment  object
{       to  be pushed. It is either the name of a standard environment object
{       or an SCL environment variable.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$push_environment
   (    object: clt$environment_object;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$environment_object
*copyc ost$status
?? POP ??
```

9.1.8 clp$pop_environment

```
{
{   This request pops the current instance of an SCL "environment object" thus
{ restoring the previously pushed instance of that object.
{
{       CLP$POP_ENVIRONMENT (OBJECT, STATUS)
{
{ OBJECT:  (input) This parameter specifies the name of the environment object
{       to be popped.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$pop_environment
   (    object: clt$environment_object;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
```

```
*copyc clt$environment_object
*copyc ost$status
?? POP ??
```

9.1.9 clp$set_lock_variable (*)

```
{
{   This request is used to set a variable of type lock.
{
{       CLP$SET_LOCK_VARIABLE (REFERENCE, WAIT, STATUS)
{
{ REFERENCE: (input)  This parameter specifies the lock variable to be set.
{
{ WAIT: (input)  This parameter specifies whether the request should wait
{       until the lock can be set or return immediately if the lock is already
{       set.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: cle$lock_already_set
{                   cle$lock_set_by_current_task
{                   cle$lock_expired
{

 PROCEDURE [XREF] clp$set_lock_variable
   (    reference: clt$variable_ref_expression;
        wait: ost$wait;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$variable_ref_expression
*copyc ost$status
*copyc ost$wait
?? POP ??
```

9.1.10 clp$clear_lock_variable (*)

```
{
{   This request is used to clear a variable of type lock. It can be used to
{ clear an expired lock. A lock which has not expired may only be cleared by
{ the task which set it.
{
{       CLP$CLEAR_LOCK_VARIABLE (REFERENCE, STATUS)
{
{ REFERENCE: (input)  This parameter specifies the lock variable to be
{       cleared.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: cle$lock_already_clear
{                   cle$lock_not_set_by_this_task
{

 PROCEDURE [XREF] clp$clear_lock_variable
   (    reference: clt$variable_ref_expression;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$variable_ref_expression
*copyc ost$status
```

```
?? POP ??
```

9.2 Command Language Variable Type Declarations

9.2.1 clt$data_access_mode

```
 TYPE
   clt$data_access_mode = (clc$read_write, clc$read_only);
```

9.2.2 clt$environment_object

```
 TYPE
   clt$environment_object = ost$name;

 { An environment object is either one of the items below
 { or an SCL environment variable.

?? FMT (FORMAT := OFF) ??

 CONST
   clc$attach_file_defaults        = 'ATTACH_FILE_DEFAULTS           ',
   clc$command_list                = 'COMMAND_LIST                   ',
   clc$file_attribute_defaults     = 'FILE_ATTRIBUTE_DEFAULTS        ',
   clc$file_connections            = 'FILE_CONNECTIONS               ',
   clc$interaction_style           = 'INTERACTION_STYLE              ',
   clc$link_attributes             = 'LINK_ATTRIBUTES                ',
   clc$message_level               = 'MESSAGE_LEVEL                  ',
   clc$natural_language            = 'NATURAL_LANGUAGE               ',
   clc$program_attributes          = 'PROGRAM_ATTRIBUTES             ',
   clc$unseen_mail_action          = 'UNSEEN_MAIL_ACTION             ',
   clc$working_catalog             = 'WORKING_CATALOG                ';

?? FMT (FORMAT := ON) ??

*copyc ost$name
```

9.2.3 clt$environment_variable_scope

```
 TYPE
   clt$environment_variable_scope = clc$push_scope .. clc$job_scope;

*copyc clt$variable_declaration_scope
```

9.2.4 clt$procedure_variable_scope

```
 TYPE
   clt$procedure_variable_scope = clc$local_scope .. clc$xref_scope;

*copyc clt$variable_declaration_scope
```

9.2.5 clt$variable_class

```
 TYPE
   clt$variable_class = (clc$library_variable, clc$environment_variable,
```

```
        clc$procedure_variable, clc$parameter_variable);
```

9.2.6 clt$variable_name

```
 TYPE
   clt$variable_name = ost$name;

*copyc ost$name
```

9.2.7 clt$variable_name_reference

```
 TYPE
   clt$variable_name_reference = string ( * <= osc$max_name_size);

*copyc ost$name
```

9.2.8 clt$variable_declaration_scope

```
 TYPE
   clt$variable_declaration_scope = (clc$local_scope, clc$xdcl_scope,
         clc$xref_scope, clc$push_scope, clc$environment_scope,
         clc$utility_scope, clc$task_scope, clc$job_scope, clc$library_scope);
```

10 Date and Time Services

10.1 Date and Time Service Interfaces

10.1.1 pmp$get_compact_date_time

```
{
{    The purpose of this request is to obtain the current date and
{  time in a compact form.
{
{        PMP$GET_COMPACT_DATE_TIME (DATE_TIME, STATUS)
{
{  DATE_TIME: (output) This parameter specifies the current date
{        and time.
{
{  STATUS: (output) This parameter specifies the request status.
{       CONDITION: pme$computed_year_out_of_range.
{       IDENTIFIER: pmc$program_management_id.
{

 PROCEDURE [XREF] pmp$get_compact_date_time (VAR date_time: ost$date_time;
   VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc OST$DATE_TIME
*copyc OST$STATUS
?? POP ??
```

10.1.2 pmp$get_universal_date_time

```
{
{   The purpose of this request is to get the current "universal" date and
{ time. The "universal" time zone is that of Greenwich, England, often called
{ Greenwich Mean Time.
{
{       PMP$GET_UNIVERSAL_DATE_TIME (UNIVERSAL_DATE_TIME, STATUS)
{
{ UNIVERSAL_DATE_TIME: (output)  This parameter specifies the universal
{       date and time.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: pme$compute_year_out_of_range
{

 PROCEDURE [XREF] pmp$get_universal_date_time
   (VAR universal_date_time: ost$date_time;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$date_time
*copyc ost$status
?? POP ??
```

10.1.3 pmp$get_day_of_week

```
{
{   The purpose of this request is to return the current day of the week.
{
{       PMP$GET_DAY_OF_WEEK (DAY_OF_WEEK, STATUS)
{
{ DAY_OF_WEEK: (output)  This parameter specifies the day of the week.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] pmp$get_day_of_week
   (VAR day_of_week: ost$day_of_week;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$day_of_week
*copyc ost$status
?? POP ??
```

10.1.4 pmp$get_time_zone

```
{
{   The purpose of this request is to return the time zone currently defined
{ for the job. Requests that return dates and times do so relative to this
{ time zone.
{
{       PMP$GET_TIME_ZONE (TIME_ZONE, STATUS)
{
{ TIME_ZONE: (output)  This parameter specifies the time zone.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] pmp$get_time_zone
   (VAR time_zone: ost$time_zone;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$time_zone
?? POP ??
```

10.1.5 pmp$compute_date_time

```
{
{    The purpose of this request is to compute a new date and time from a base
{  date and time, and an increment.
{
{        PMP$COMPUTE_DATE_TIME (BASE, INCREMENT, RESULT, STATUS)
{
{ BASE: (input) This parameter specifies the date and time to increment.
{
{ INCREMENT: (input) This parameter specifies the amount to increment the base
{        date and time by, either positive or negative.
```

```
{
{ RESULT: (output) This parameter specifies the resulting date and time.
{
{ STATUS: (output) This parameter specifies the request status.
{    CONDITION: pme$compute_overflow, pme$invalid_year.
{    IDENTIFIER: pmc$program_management_id.
{

 PROCEDURE [XREF] pmp$compute_date_time (base: ost$date_time;
       increment: pmt$time_increment;
   VAR result: ost$date_time;
   VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc OST$DATE_TIME
*copyc PMT$TIME_INCREMENT
*copyc OST$STATUS
?? POP ??
```

10.1.6 pmp$compute_date_time_increment

```
{
{   The purpose of this request is to compute the time increment between an old
{ date and time and a new date and time.
{
{       PMP$COMPUTE_DATE_TIME_INCREMENT (OLD, NEW, INCREMENT, STATUS)
{
{ OLD: (input)  This parameter specifies the first date and time.
{
{ NEW: (input)  This parameter specifies the second date and time. If OLD is
{       earlier in time than NEW, then INCREMENT will be positive.
{
{ INCREMENT: (output)  This parameter specifies the resulting date and time
{       increment, either positive or negative.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: pme$compute_overflow
{                   pme$invalid_year
{

 PROCEDURE [XREF] pmp$compute_date_time_increment (old: ost$date_time;
       new: ost$date_time;
   VAR increment: pmt$time_increment;
   VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc OST$DATE_TIME
*copyc PMT$TIME_INCREMENT
*copyc OST$STATUS
?? POP ??
```

10.1.7 pmp$compute_local_date_time

```
{
{   The purpose of this request is to compute the "local" date and time
{ equivalent to a date and time expressed relative to the "universal" time
{ zone. The "universal" time zone is that of Greenwich, England, often called
{ Greenwich Mean Time.
```

```
{
{       PMP$COMPUTE_LOCAL_DATE_TIME (UNIVERSAL_DATE_TIME, TIME_ZONE,
{         LOCAL_DATE_TIME, STATUS)
{
{ UNIVERSAL_DATE_TIME: (input)  This parameter specifies the universal date
{       and time.
{
{ TIME_ZONE: (input)  This parameter specifies the time zone that the date and
{       time are to be made relative to.
{
{ LOCAL_DATE_TIME: (output)  This parameter specifies the resulting local date
{       and time.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: pme$compute_overflow
{                   pme$invalid_year
{                   pme$invalid_time_zone
{

 PROCEDURE [XREF] pmp$compute_local_date_time
   (    universal_date_time: ost$date_time;
        time_zone: ost$time_zone;
    VAR local_date_time: ost$date_time;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$date_time
*copyc ost$status
*copyc ost$time_zone
?? POP ??
```

10.1.8 pmp$compute_universal_date_time

```
{
{   The purpose of this request is to compute the "universal" date and time
{ equivalent to a date and time expressed relative to a "local" time zone.
{ The "universal" time zone is that of Greenwich, England, often called
{ Greenwich Mean Time.
{
{       PMP$COMPUTE_UNIVERSAL_DATE_TIME (LOCAL_DATE_TIME, TIME_ZONE,
{         UNIVERSAL_DATE_TIME, STATUS)
{
{ LOCAL_DATE_TIME: (input)  This parameter specifies the local date and time.
{
{ TIME_ZONE: (input)  This parameter specifies the time zone that the local
{       date and time is relative to.
{
{ UNIVERSAL_DATE_TIME: (output)  This parameter specifies the resulting
{       universal date and time.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: pme$compute_overflow
{                   pme$invalid_year
{                   pme$invalid_time_zone
{

 PROCEDURE [XREF] pmp$compute_universal_date_time
   (    local_date_time: ost$date_time;
        time_zone: ost$time_zone;
```

```
    VAR universal_date_time: ost$date_time;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$date_time
*copyc ost$status
*copyc ost$time_zone
?? POP ??
```

10.1.9 pmp$compute_day_of_week

```
{
{   The purpose of this request is to determine the day of the week
{ corresponding to a date.
{
{       PMP$COMPUTE_DAY_OF_WEEK (DATE, DAY_OF_WEEK, STATUS)
{
{ DATE: (input)  This parameter specifies the date for which the day of the
{       week is desired.
{
{ DAY_OF_WEEK: (output)  This parameter specifies the day of the week.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: pme$invalid_year
{

 PROCEDURE [XREF] pmp$compute_day_of_week
   (    date: ost$date_time;
    VAR day_of_week: ost$day_of_week;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$date_time
*copyc ost$day_of_week
*copyc ost$status
?? POP ??
```

10.1.10 clp$get_date_string

```
{
{   This request produces the string representation of the current date in a
{ site defined format.
{
{       CLP$GET_DATE_STRING (STR, STATUS)
{
{ STR: (output)  This parameter specifies the result string.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] clp$get_date_string
   (VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$string
```

```
?? POP ??
```

10.1.11 clp$get_date_time_string

```
{
{   This request produces the string representation of the current date and/or
{ time and/or day of the week.
{
{       CLP$GET_DATE_TIME_STRING (FORMAT, STR, STATUS)
{
{ FORMAT: (input)  This parameter specifies the "date time form string" to be
{       used to guide the construction of the representation. If this string
{       is null or all spaces a site defined default is used.
{
{ STR: (output)  This parameter specifies the result string.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$get_date_time_string
   (    format: clt$date_time_form_string;
    VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$date_time_form_string
*copyc ost$status
*copyc ost$string
?? POP ??
```

10.1.12 clp$get_day_name

```
{
{   The purpose of this request is to get the name of the specified day of the
{ week. These names are kept on "month and day name" message modules (on an
{ object library in the command list).
{
{       CLP$GET_DAY_NAME (DAY_OF_WEEK, FULL_FORM, NATURAL_LANGUAGE, DAY_NAME,
{         STATUS
{
{ DAY_OF_WEEK: (input)  This parameter specifies the day of the week whose
{       name is desired.
{
{ FULL_FORM: (input)  This parameter specifies whether the "full" form of the
{       name is to be returned (TRUE) or the name's abbreviation (FALSE).
{
{ NATURAL_LANGUAGE: (input)  This parameter specifies the natural language in
{       which the name is to be returned. Osc$null_name can be used to
{       specify the job's currently selected natural language. If no
{       definition for the day can be found in the requested natural language,
{       US_English is used.
{
{ DAY_NAME: (output)  This parameter specifies the returned day name.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: ose$bad_day_of_week
{                   ose$bad_natural_language
```

```
{

 PROCEDURE [XREF] clp$get_day_name
   (    day_of_week: ost$day_of_week;
        full_form: boolean;
        natural_language: ost$natural_language;
    VAR day_name: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$day_of_week
*copyc ost$natural_language
*copyc ost$status
*copyc ost$string
?? POP ??
```

10.1.13 clp$get_month_name

```
{
{   The purpose of this request is to get the name of the specified month
{ These names are kept on "month and day name" message modules (on an object
{ library in the command list).
{
{       CLP$GET_MONTH_NAME (MONTH_NUMBER, FULL_FORM, NATURAL_LANGUAGE,
{         MONTH_NAME, STATUS
{
{ MONTH_NUMBER: (input)  This parameter specifies the number (1..12) of the
{       month whose name is desired.
{
{ FULL_FORM: (input)  This parameter specifies whether the "full" form of the
{       name is to be returned (TRUE) or the name's abbreviation (FALSE).
{
{ NATURAL_LANGUAGE: (input)  This parameter specifies the natural language in
{       which the name is to be returned. Osc$null_name can be used to
{       specify the job's currently selected natural language. If no
{       definition for the month can be found in the requested natural
{       language, US_English is used.
{
{ MONTH_NAME: (output)  This parameter specifies the returned month name.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$bad_month_number
{                   ose$bad_natural_language
{

 PROCEDURE [XREF] clp$get_month_name
   (    month_number: 1 .. 12;
        full_form: boolean;
        natural_language: ost$natural_language;
    VAR month_name: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$natural_language
*copyc ost$status
*copyc ost$string
?? POP ??
```

10.1.14 clp$get_time_string

```
{
{   This request produces the string representation of the current time in a
{ site defined format.
{
{       CLP$GET_TIME_STRING (STR, STATUS)
{
{ STR: (output)  This parameter specifies the result string.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] clp$get_time_string
   (VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$string
?? POP ??
```

10.1.15 clp$get_time_zone_identifier

```
{
{   The purpose of this request is to get the identifier of the specified time
{ zone. These identifiers are kept on "time zone message modules" (on an
{ object library in the command list).
{
{       CLP$GET_TIME_ZONE_IDENTIFIER (TIME_ZONE, FULL_FORM, NATURAL_LANGUAGE,
{         TIME_ZONE_IDENTIFIER, STATUS)
{
{ TIME_ZONE: (input)  This parameter specifies the time zone whose identifier
{       is desired.
{
{ FULL_FORM: (input)  This parameter specifies whether the "full" form of the
{       identifier is to be returned (TRUE) or the identifier's abbreviation
{       (FALSE).
{
{ NATURAL_LANGUAGE: (input)  This parameter specifies the natural language in
{       which the identifier is to be returned. Osc$null_name can be used to
{       specify the job's currently selected natural language. If no
{       definition for the time zone can be found in the requested natural
{       language, US_English is used. If a definition still cannot be found,
{       a null string (size = 0) is returned.
{
{ TIME_ZONE_IDENTIFIER: (output)  This parameter specifies the returned time
{       zone identifier.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: pme$invalid_time_zone
{                   ose$bad_natural_language
{

 PROCEDURE [XREF] clp$get_time_zone_identifier
   (    time_zone: ost$time_zone;
        full_form: boolean;
        natural_language: ost$natural_language;
```

```
    VAR time_zone_identifier: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$natural_language
*copyc ost$status
*copyc ost$string
*copyc ost$time_zone
?? POP ??
```

10.2 Date and Time Services Type Declarations

10.2.1 clt$date_time

```
TYPE
  clt$date_time = record
    value: ost$date_time,
    date_specified: boolean,
    time_specified: boolean,
  recend;

*copyc ost$date_time
```

10.2.2 clt$date_time_form_string

```
TYPE
  clt$date_time_form_string = string ( * <= clc$max_date_time_form_string);

*copyc clc$max_date_time_form_string
```

10.2.3 clc$max_date_time_form_string

```
CONST
  clc$max_date_time_form_string = 255;
```

10.2.4 ost$date_time

```
TYPE
  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;
```

10.2.5 ost$day_of_week

```
TYPE
  ost$day_of_week = (osc$monday, osc$tuesday, osc$wednesday, osc$thursday,
        osc$friday, osc$saturday, osc$sunday);
```

10.2.6 ost$time_zone

```
TYPE
  ost$time_zone = record
    hours_from_gmt: -12 .. 12,
    minutes_offset: -30 .. 30,
    daylight_saving_time: boolean,
  recend;
```

10.2.7 pmt$time_increment

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

11 Miscellaneous Services

11.1 Miscellaneous Services Interfaces

11.1.1 clp$count_list_elements

```
{
{   This function returns a count of the number of elements in an SCL list
{ value. If the list is empty, zero is returned. If the value passed to
{ this function is not a properly formed list the results are undefined.
{
{       CLP$COUNT_LIST_ELEMENTS (LIST_VALUE):  COUNT
{
{ LIST_VALUE: (input)  This parameter specifies the list value whose elements
{       are to be counted.
{

 FUNCTION [INLINE] clp$count_list_elements
   (    list_value: ^clt$data_value): clt$list_size;

?? PUSH (LISTEXT := ON) ??
*copyc clt$list_size
?? POP ??
```

11.1.2 clp$evaluate_expression

```
{
{   This request evaluates an SCL expression.
{
{       CLP$EVALUATE_EXPRESSION (EXPRESSION, TYPE_SPECIFICATION, WORK_AREA,
{         RESULT, STATUS)
{
{ EXPRESSION: (input)  This parameter specifies the expression to be
{       evaluated.
{
{ TYPE_SPECIFICATION: (input)  This parameter specifies the data type the
{       expression is expected to result in.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       will be used to construct the expression's result value. The current
{       position of this sequence pointer is updated to reflect the amount of
{       storage used by the request. The expression's result value is
{       completely contained within the used part of this sequence.
{
{ RESULT: (output)  This parameter specifies the expression's result value.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$evaluate_expression
   (    expression: clt$expression_text;
        type_specification: ^clt$type_specification;
    VAR work_area {input, output} : ^clt$work_area;
    VAR result: ^clt$data_value;
```

```
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$data_value
*copyc clt$expression_text
*copyc clt$type_specification
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

11.1.3 clp$evaluate_expression_to_str

```
{
{   This request evaluates an SCL expression using a type specification of ANY
{ and returns the result in its string representation. Also returned is the
{ name of the generic type of the result.
{
{       CLP$EVALUATE_EXPRESSION_TO_STR (EXPRESSION, RESULT_STRING, TYPE_NAME,
{         STATUS)
{
{ EXPRESSION: (input)  This parameter specifies the expression to be
{       evaluated.
{
{ RESULT_STRING: (output)  This parameter specifies the string representation
{       of the expression's result value. The string must be long enough to
{       hold the result's representation.
{
{ TYPE_NAME: (output)  This parameter specifies the name of the generic type
{       of the result.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:  cle$string_too_short
{

 PROCEDURE [XREF] clp$evaluate_expression_to_str
   (    expression: clt$expression_text;
    VAR result_string: clt$string_value;
    VAR type_name: clt$type_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$expression_text
*copyc clt$string_value
*copyc clt$type_name
*copyc ost$status
?? POP ??
```

11.1.4 clp$evaluate_token

```
{
{   The purpose of this request is to scan the next lexical token in a line.
{ On entry, INDEX indicates where to begin scanning TEXT. On exit, INDEX
{ indicates where scanning stopped (i.e. the next character, if any, to be
{ scanned).
{
{   The TEXT_INDEX field of TOKEN is set to the position of the first
{ character of the returned token. The TEXT_SIZE field of TOKEN is set to the
{ number of characters occupied by the token in the text (i.e. exit value of
```

```
{ INDEX - TOKEN.TEXT_INDEX). The DESCRIPTOR field of TOKEN is set to a string
{ that describes the token, e.g. for use in error messages.
{
{   Name tokens are 1 to 31 characters in length and are returned in TOKEN.STR
{ with all lower case letters translated to their upper case counterparts.
{ The following BNF definitions illustrate the syntax of a name:
{
{    <clc$name_token> ::= <alphabetic char> [<alphanumeric char>]...
{    <alphanumeric char> ::= <alphabetic char> | <digit>
{    <alphabetic char> ::= <letter>
{                        | <special alphabetic char>
{                        | <international letter>
{    <letter> ::= <upper case letter> | <lower case letter>
{    <upper case letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M
{                          | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
{    <lower case letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
{                          | n | o | p | q | r | s | t | u | v | w | x | y | z
{    <international letter> ::= <upper case international letter>
{                             | <lower case international letter>
{    <upper case international letter> ::= @ | '[' |  \  | ^ | ']'
{    <lower case international letter> ::= ` |  {  | '|' | ~ |  }
{    <special alphabetic char> ::= # | $ | _
{    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
{
{   A CYBIL name token differs from an SCL name token in that it must start
{ with a letter and may not contain any of the international alphabetic
{ characters except @. The distinction between a CYBIL name and an SCL name
{ is made only if the scan option CLC$CLASSIFY_NAME_TOKEN is selected. The
{ following BNF definitions illustrate the syntax of a CYBIL name:
{
{    <clc$cybil_name_token> ::= <letter> [<cybil name char>]...
{    <cybil name char> ::= <letter>
{                        | <special alphabetic char>
{                        | @
{                        | <digit>
{
{   A special CYBIL name token differs from a regular CYBIL name token in that
{ it starts with either a $ or a # and is 2 to 32 characters in length. The
{ intent of this kind of token is to accomodate set value constructor
{ notation. A special CYBIL name is recognized as such only if the scan
{ options CLC$CLASSIFY_NAME_TOKEN and CLC$SPECIAL_CYBIL_NAME_IS_TOKEN are
{ selected. The following BNF definitions illustrate the syntax of a special
{ CYBIL name:
{
{    <clc$special_cybil_name_token> ::= $ <clc$cybil_name_token>
{                                     | # <clc$cybil_name_token>
{
{   A simple name token contains only letters or digits and must start with a
{ letter. Thus a simple name may be considered to be any of the other kinds
{ of names (except a special CYBIL name). The distinction between a simple
{ name and an SCL name is made only if the scan option CLC$CLASSIFY_NAME_TOKEN
{ is selected. The following BNF definitions illustrate the syntax of a
{ simple name.
{
{    <clc$simple_name_token> ::= <letter> [<letter | digit>]...
{    <letter | digit> ::= <letter> | <digit>
{
{   COBOL name tokens are 1 to 30 characters in length and may contain
{ hyphens. A COBOL name is recognized only if the scan options
{ CLC$CLASSIFY_NAME_TOKEN and CLC$COBOL_NAME_IS_TOKEN are selected. The
```

```
{ following BNF definitions illustrate the syntax of a COBOL name:
{
{    <clc$cobol_name_token> ::=
{          <letter> [[<COBOL name char>]... <letter | digit>]
{        | <digit> [<COBOL name char>]... <letter>
{                 [[<COBOL name char>]... <letter | digit>]
{    <COBOL name char> ::= <letter> | <digit> | <hyphen>
{    <letter | digit> ::= <letter> | <digit>
{    <hyphen> ::= -
{
{   Integer tokens may have radix specifications and must be delimited at both
{ ends. The default radix is decimal (10). The following BNF definitions
{ illustrate the syntax of integers:
{
{    <clc$signed_integer_token> ::= <sign> <clc$unsigned_integer_token>
{    <sign> ::= + | -
{    <clc$unsigned_integer_token> ::=  <digit> [<hex digit>]... [(<radix>)]
{    <hex digit> ::= <digit> | A | B | C | D | E | F
{                             | a | b | c | d | e | f
{    <radix> ::= <unsigned decimal>
{    <unsigned decimal> ::= <digit>...
{
{   Real tokens, must have fractional portions and may have an exponent. The
{ following BNF definitions illustrate the syntax of real numbers:
{
{    <clc$signed_real_token> ::= <sign> <clc$unsigned_real_token>
{    <clc$unsigned_real_token> ::= <mantissa> [<exponent>]
{    <mantissa> ::= <integer part> . <fraction part>
{    <integer part> ::= <unsigned decimal>
{    <fraction part> ::= <unsigned decimal>
{    <exponent> ::= <E|e> [<sign>] <unsigned decimal>
{    <E|e> ::= E | e
{
{   Strings must be enclosed in apostrophes (single quote marks). In the STR
{ field of TOKEN, the enclosing apostrophes are removed and doubled
{ apostrophes within the original TEXT are replaced by a single apostrophe.
{ The token can hold a string up to 256 characters long. If the STR_COMPLETE
{ field of TOKEN is FALSE, the string is too large to fit in TOKEN and must be
{ accessed via the TEXT_INDEX and TEXT_SIZE fields. The following BNF
{ definitions illustrate the syntax of strings:
{
{    <clc$string_token> ::= ' [<string char>]... '
{    <string char> ::= <any ascii character except '> | ''
{
{   Comments are treated as spaces unless the scan option CLC$COMMENT_IS_TOKEN
{ is selected. The following BNF definitions illustrate the syntax of
{ comments:
{
{    <clc$comment_token> ::= " [<comment char>]... ["]
{    <comment char> ::= <any ascii character except ">
{
{   The following BNF definitions illustrate the representation of the
{ delimiter and operator tokens:
{
{    <clc$semicolon_token> ::= ;
{    <clc$colon_token> ::= :
{    <clc$cybil_assign_token> ::= :=
{    <clc$left_parenthesis_token> ::= (
{    <clc$right_parenthesis_token> ::= )
{    <clc$comma_token> ::= ,
```

```
{    <clc$ellipsis_token> ::= .. [.]...
{    <clc$dot_token> ::= .
{    <clc$query_token> ::= ?
{    <clc$greater_than_token> ::= >
{    <clc$greater_equal_token> ::= >=
{    <clc$less_than_token> ::= <
{    <clc$less_equal_token> ::= <=
{    <clc$equal_token> ::= =
{    <clc$not_equal_token> ::= <>
{    <clc$assign_token> ::= =
{    <clc$concatenate_token> ::= //
{    <clc$exponentiate_token> ::= **
{    <clc$multiply_token> ::= *
{    <clc$divide_token> ::= /
{    <clc$add_token> ::= +
{    <clc$subtract_token> ::= -
{
{   The following are considered tokens only if the scan options
{ CLC$CLASSIFY_NAME_TOKEN and CLC$INTERNATIONAL_CHAR_IS_TOKEN are selected.
{
{    <clc$commercial_at_token> ::= @
{    <clc$left_bracket_token> ::= '['
{    <clc$reverse_slant_token> ::= \
{    <clc$right_bracket_token> ::= ']'
{    <clc$circumflex_token> ::= ^
{    <clc$grave_accent_token> ::= `
{    <clc$left_brace_token> ::= {
{    <clc$vertical_bar_token> ::= '|'
{    <clc$right_brace_token> ::= }
{    <clc$tilde_token> ::= ~
{
{   The following are considered tokens only if the scan options
{ CLC$CLASSIFY_NAME_TOKEN and CLC$SPECIAL_CHAR_IS_TOKEN are selected.
{
{    <clc$number_sign_token> ::= #
{    <clc$dollar_sign_token> ::= $
{    <clc$underscore_token> ::= _
{
{   Contiguous spaces are treated collectively as a clc$space_token. The
{ horizontal tab (HT) character is treated identically to the space character.
{ Also, comments are normally treated as spaces.
{
{   Any character that does not begin a token previously described, is
{ returned as a clc$unknown_token.
{
{
{       CLP$EVALUATE_TOKEN (TEXT, SCAN_OPTIONS, INDEX, SPACES_PRECEDED_TOKEN,
{         TOKEN, STATUS)
{
{ TEXT: (input)  This parameter specifies the text to be scanned.
{
{ SCAN_OPTIONS: (input)  This parameter specifies options for the token
{       scanning process. These options are:
{
{       clc$ignore_spaces_before_token:  if selected and a clc$space_token
{             would be returned, the space token is bypassed and the next
{             token is returned. For a clc$signed_integer_token or a
{             clc$signed_real_tokens, this applies to spaces appearing between
{             the sign and the number as well.
{
```

```
{       clc$comment_is_token:  if selected this option defeats the normal
{             process of comments being considered as equivalent to spaces.
{
{       clc$classify_name_token:  if selected this option causes names to be
{             classified according to the BNF definitions above.
{
{       clc$cobol_name_is_token:  if selected this option allows a
{             clc$cobol_name_token to be returned. This option is ignored
{             unless clc$classify_name_token is also selected.
{
{       clc$special_cybil_name_is_token:  if selected this option allows a
{             clc$special_cybil_name_token to be returned. This option is
{             ignored unless clc$classify_name_token is also selected.
{
{       clc$international_char_is_token:  if selected allows the international
{             characters, normally considered to be part of a name, to be
{             returned as tokens (see above). This option is ignored unless
{             clc$classify_name_token is also selected.
{
{       clc$special_char_is_token:  if selected allows the special characters,
{             normally considered to be part of a name, to be returned as
{             tokens (see above). This option is ignored unless
{             clc$classify_name_token is also selected.
{
{ INDEX: (input, output)  This parameter specifies the next character within
{       TEXT to be scanned.
{
{ SPACES_PRECEDED_TOKEN: (output)  This parameter specifies whether spaces
{       were found before the returned token and is only meaningful if the
{       clc$ignore_spaces_before_token option is selected.
{
{ TOKEN: (output)  This parameter specifes the returned token.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$evaluate_token
   (    text: clt$string_value;
        evaluation_options: clt$token_evaluation_options;
    VAR index {input, output} : clt$string_index;
    VAR spaces_preceded_token: boolean;
    VAR token: clt$lexical_token;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_lexical
*copyc clt$lexical_token
*copyc clt$string_index
*copyc clt$string_value
*copyc clt$token_evaluation_options
*copyc ost$status
?? POP ??
```

11.1.5 clp$execute_command

```
{
{   The purpose of this request is to execute a command asynchronously in a
{ new task. Any command other than a utility subcommand may be executed via
```

```
{ this request.
{
{   If the command being executed is not already packaged as a program, i.e.
{ it is an SCL procedure or built_in system command, an internal SCL program
{ is executed which in turn calls the command. Otherwise, the command's own
{ program description is used.
{
{       CLP$EXECUTE_COMMAND (COMMAND, COMMAND_FILE, ENABLE_ECHOING, TASK_NAME,
{         TASK_ID, STATUS)
{
{ COMMAND: (input)  This parameter specifies the command to be executed. The
{       text of this string must conform to the syntax of a single command.
{
{ COMMAND_FILE: (input)  This parameter specifies a file which becomes the
{       current command file (i.e. $COMMAND) within the new task. This
{       parameter is only meaningful if the command being executed is a
{       command utility, or for some other reason references the current
{       command file. A null or blank string can be used to specify the
{       absence of a command file.
{
{ ENABLE_ECHOING: (input)  This  parameter determines whether the command may
{       be echoed (TRUE) or not (FALSE).
{
{ TASK_NAME: (input)  This parameter specifies the name to be used to refer to
{       the task subsequent to this request (e.g. in a clp$get_task_status
{       request).
{
{ TASK_ID: (output)  This parameter specifies the identifier assigned to the
{       task and can be used in subsequent program management requests to
{       refer to the task.
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$execute_command
   (    command: clt$command_line;
        command_file: fst$file_reference;
        enable_echoing: boolean;
        task_name: clt$task_name_reference;
    VAR task_id: pmt$task_id;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_command_processing
*copyc clt$command_line
*copyc clt$task_name_reference
*copyc fst$file_reference
*copyc ost$status
*copyc pmt$task_id
?? POP ??
```

11.1.6 clp$generate_pdt

```
{
{   The purpose of this request is to produce a Parameter Description Table
{ (PDT) from input in the form of an SCL procedure declaration. In addition
{ to the FIRST_LINE to be processed, input to this request may be provided
{ dynamically by supplying an input procedure that obtains subsequent input as
{ required by this interface.
```

```
{
{   This interface assumes that the first word of the input declaration has
{ been processed by the caller; i.e. whether the declaration begins with
{ PROCEDURE or FUNCTION has been determined and used to establish the value of
{ the COMMAND_OR_FUNCTION parameter.
{
{       CLP$GENERATE_PDT (COMMAND_OR_FUNCTION, FIRST_LINE, FIRST_LINE_INDEX,
{         GET_LINE, WORK_AREA, LAST_LINE, LAST_LINE_INDEX,
{         COMMAND_OR_FUNCTION_NAME, ALIASES, AVAILABILITY,
{         COMMAND_OR_FUNCTION_SCOPE, COMMAND_LOG_OPTION,
{         PARAMETER_DESCRIPTION_TABLE, STATUS)
{
{ COMMAND_OR_FUNCTION: (input)  This parameter specifies whether the PDT being
{       generated is for a command or a function.
{
{ FIRST_LINE: (input)  This parameter specifies the first (possibly only) line
{       to be processed by this request.
{
{ FIRST_LINE_INDEX: (input)  This parameter specifies the position within the
{       first line at which processing is to begin.
{
{ GET_LINE: (input)  This parameter specifies the procedure that this request
{       calls to get its input dynamically. The procedure is responsible for
{       concatenating continuation lines to form a "command line". NIL may be
{       specified to indicate the absence of an input procedure.
{
{             input_procedure (LINE, STATUS)
{
{       LINE: (output)  This parameter specifies the line. A NIL pointer is
{             used to indicate end of input.
{
{       STATUS: (output)  This parameter specifies the input procedure's
{             completion status.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       will be used to construct the PDT and aliases. The current position
{       of this sequence pointer is updated to reflect the amount of storage
{       used by the request. The PDT and aliases are completely contained
{       within the used part of this sequence.
{
{ LAST_LINE: (output)  This parameter specifies the last line processed by
{       this interface. If an input procedure is supplied and is used by the
{       request, this is set to the last line that procedure returned;
{       otherwise this is set to FIRST_LINE.
{
{ LAST_LINE_INDEX: (output)  This parameter specifies the position within
{       LAST_LINE that immediately follows the last character that was
{       processed.
{
{ COMMAND_OR_FUNCTION_NAME: (output)  This parameter specifies the name of the
{       command or function for which the PDT is being generated.
{
{ ALIASES: (output)  This parameter specifies the aliases for the command or
{       function name. NIL indicates there are no aliases.
{
{ AVAILABILITY: (output)  This parameter specifies whether the command or
{       function is to be advertised or hidden (for instance, by the
{       display_command_list_entry command).
{
{ COMMAND_OR_FUNCTION_SCOPE: (output)  This parameter specifies the scope of
```

```
{       the command or function.
{
{ COMMAND_LOG_OPTION: (output)  This parameter specifies whether calls to the
{       command are to be logged automatically or manually.
{
{ PARAMETER_DESCRIPTION_TABLE: (output)  This parameter specifies the
{       generated PDT.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$work_area_overflow
{

 PROCEDURE [XREF] clp$generate_pdt
   (    command_or_function: clt$command_or_function;
        first_line: ^clt$command_line;
        first_line_index: clt$command_line_index;
        get_line: clt$input_procedure;
    VAR work_area {input, output} : ^clt$work_area;
    VAR last_line: ^clt$command_line;
    VAR last_line_index: clt$command_line_index;
    VAR command_or_function_name: pmt$program_name;
    VAR aliases: ^array [1 .. * ] of pmt$program_name;
    VAR availability: clt$named_entry_availability;
    VAR command_or_function_scope: clt$command_or_function_scope;
    VAR command_log_option: clt$command_log_option;
    VAR parameter_description_table: ^clt$parameter_description_table;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$command_line
*copyc clt$command_line_index
*copyc clt$command_log_option
*copyc clt$command_or_function
*copyc clt$command_or_function_scope
*copyc clt$input_procedure
*copyc clt$named_entry_availability
*copyc clt$parameter_description_table
*copyc clt$work_area
*copyc ost$status
*copyc pmt$program_name
?? POP ??
```

11.1.7 clp$generate_type_specification

```
{
{   The purpose of this request is to produce a clt$type_specification from
{ input in the form of an SCL type expression. In addition to the FIRST_LINE
{ to be processed, input to this request may be provided dynamically by
{ supplying an input procedure that obtains subsequent input as required by
{ this interface.
{
{       CLP$GENERATE_TYPE_SPECIFICATION (TYPE_NAME, FIRST_LINE,
{         FIRST_LINE_INDEX, GET_LINE, WORK_AREA, LAST_LINE, LAST_LINE_INDEX,
{         TYPE_SPECIFICATION, STATUS)
{
{ TYPE_NAME: (input)  This parameter specifies the name of the type for which
{       the specifiction is being generated.
{
{ FIRST_LINE: (input)  This parameter specifies the first (possibly only) line
```

```
{       to be processed by this request.
{
{ FIRST_LINE_INDEX: (input)  This parameter specifies the position within the
{       first line at which processing is to begin.
{
{ GET_LINE: (input)  This parameter specifies the procedure that this request
{       calls to get its input dynamically. The procedure is responsible for
{       concatenating continuation lines to form a "command line". NIL may be
{       specified to indicate the absence of an input procedure.
{
{             input_procedure (LINE, STATUS)
{
{       LINE: (output)  This parameter specifies the line. A NIL pointer is
{             used to indicate end of input.
{
{       STATUS: (output)  This parameter specifies the input procedure's
{             completion status.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       will be used to construct the type specification. The current
{       position of this sequence pointer is updated to reflect the amount of
{       storage used by the request. The type specification is completely
{       contained within the used part of this sequence.
{
{ LAST_LINE: (output)  This parameter specifies the last line processed by
{       this interface. If an input procedure is supplied and is used by the
{       request, this is set to the last line that procedure returned;
{       otherwise this is set to FIRST_LINE.
{
{ LAST_LINE_INDEX: (output)  This parameter specifies the position within
{       LAST_LINE that immediately follows the last character that was
{       processed.
{
{ TYPE_SPECIFICATION: (output)  This parameter specifies the generated type
{       specification.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:

 PROCEDURE [XREF] clp$generate_type_specification
   (    type_name: clt$type_name;
        first_line: ^clt$command_line;
        first_line_index: clt$command_line_index;
        get_line: clt$input_procedure;
    VAR work_area {input, output} : ^clt$work_area;
    VAR last_line: ^clt$command_line;
    VAR last_line_index: clt$command_line_index;
    VAR type_specification: ^clt$type_specification;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$command_line
*copyc clt$command_line_index
*copyc clt$input_procedure
*copyc clt$type_name
*copyc clt$type_specification
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

11.1.8 clp$get_command_origin

```
{
{   The purpose  of this request is to determine whether the command currently
{ being processed was entered directly from an interactive device.
{
{       CLP$GET_COMMAND_ORIGIN (INTERACTIVE, STATUS)
{
{ INTERACTIVE: (output) This parameter specifies whether the  current  command
{       came from an interactive device (true) or not (false).
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{       IDENTIFIER: 'CL'
{

 PROCEDURE [XREF] clp$get_command_origin
   (VAR interactive: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
?? POP ??
```

11.1.9 clp$get_source

```
{
{   The purpose of this request is to return the source (residence) of the
{ requesting command or function.
{
{       CLP$GET_SOURCE (SOURCE, STATUS)
{
{ SOURCE: (output)  This parameter specifies the source of the command or
{       function. The KIND field specifies one of:
{
{       CLC$SYSTEM_SOURCE:  the command or function is part of the system.
{
{       CLC$UTILITY_SOURCE:  the command or function is part of the utility
{             designated by the UTILITY_NAME field.
{
{       CLC$CATALOG_SOURCE:  the command resides on a file in the catalog
{             designated by the PATH_NAME field.
{
{       CLC$LIBRARY_SOURCE:  the command or function resides in the object
{             library designated by the PATH_NAME field.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_source
   (VAR source: clt$source;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$source
*copyc ost$status
?? POP ??
```

11.1.10 clp$get_task_status

```
{
{     The purpose of this request is to get the status of an asynchronous task.
{
{       CLP$GET_TASK_STATUS (TASK_NAME, TASK_STATUS, STATUS)
{
{ TASK_NAME: (input)  This parameter specifies the name of the task whose
{       status is requested.
{
{ TASK_STATUS: (output)  This parameter specifies whether the task has
{       completed and if so, what its completion status was. If a TASK_NAME
{       is specified which is unknown, this parameter will indicate that the
{       task completed normally.
{
{ STATUS: (output)  This parameter specifies the request staus.
{

 PROCEDURE [XREF] clp$get_task_status
   (    task_name: clt$task_name_reference;
    VAR task_status: pmt$task_status;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$task_name_reference
*copyc ost$status
*copyc pmt$task_status
?? POP ??
```

11.1.11 clp$get_type_information

```
{
{   This request retrieves information from a clt$type_specification. It can
{ be used, for example, to determine the minimum and maximum size for a string,
{ the bounds for an array, etc.
{
{       CLP$GET_TYPE_INFORMATION (TYPE_SPECIFICATION, WORK_AREA,
{         TYPE_INFORMATION, STATUS)
{
{ TYPE_SPECIFICATION: (input)  This parameter specifies the type specification
{       from which information is to be obtained.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage which
{       is used to hold additional information about the type that is pointed
{       to from the TYPE_INFORMATION parameter. The current position of this
{       sequence pointer is updated to reflect the amount of storage used by
{       the request.
{
{ TYPE_INFORMATION: (output)  This parameter specifies the information about
{       the type.
{
{ STATUS: (output) This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$get_type_information
   (    type_specification: ^clt$type_specification;
    VAR work_area {input, output} : ^clt$work_area;
    VAR type_information: clt$type_information;
    VAR status: ost$status);
```

```
?? PUSH (LISTEXT := ON) ??
*copyc clt$type_information
*copyc clt$type_specification
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

11.1.12 clp$include_command

```
{
{   The purpose of this request is to interpret a string as one or more
{ commands.
{
{   This request differs from the clp$include_line request in that the latter
{ builds a whole new "input control block" in which to process the commands and
{ control statements, whereas clp$include_command processes them in the context
{ of the current "input control block".
{
{       CLP$INCLUDE_COMMAND (COMMAND, ENABLE_ECHOING, STATUS)
{
{ COMMAND: (input)  This parameter specifies the command(s) to be interpreted.
{
{ ENABLE_ECHOING: (input)  This parameter determines whether the command may be
{       echoed (TRUE) or not (FALSE).
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] clp$include_command
   (    command: clt$command_line;
        enable_echoing: boolean;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$command_line
*copyc ost$status
?? POP ??
```

11.1.13 clp$substitute_delimited_text

```
{
{   This request scans for delimited SCL string expressions within a  line  of
{ text.  Each  expression  is evaluated and the result replaces the delimited
{ text including the delimiters.
{
{   If a second delimiter is not found in a line, the end-of-line  is  treated
{ as  the  second  delimiter.  Two consecutive delimiters are replaced with a
{ single delimiter.
{
{   If an expression cannot be  evaluated  the  original  text  including  the
{ delimiters  is  not  replaced.  If the new line of text exceeds the maximum
{ line size, the original line  is  returned.  In  both  cases,  the  request
{ terminates with an appropriate status.
{
{       CLP$SUBSTITUTE_DELIMITED_TEXT (OLD_TEXT, DELIMITER, NEW_TEXT,
{         NEW_TEXT_SIZE, STATUS)
{
```

```
{ OLD_TEXT: (input) This parameter specifies the text to be processed.
{
{ DELIMITER:  (input)  This  parameter  specifies the delimiter for the string
{       expressions.
{
{ NEW_TEXT: (output) This parameter specifies the result of  the  substitution
{       process.
{
{ NEW_TEXT_SIZE: (output) This parameter specifies the length of the result of
{       the substitution process.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$substitute_delimited_text
   (    old_text: clt$command_line;
        delimiter: char;
    VAR new_text: clt$command_line;
    VAR new_text_size: clt$command_line_size;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$command_line
*copyc clt$command_line_size
*copyc ost$status
?? POP ??
```

11.1.14 clp$trimmed_string_size

```
{
{   The purpose  of  this  function  is  to  return  the size of a string once
{ trailing space characters have been removed from  it.  The  horizontal  tab
{ (HT)  in addition to the space are considered to be space characters by this
{ function.
{
{       CLP$TRIMMED_STRING_SIZE (STR): TRIMMED_STRING_SIZE
{
{ STR: (input) This parameter specifies the string for which the trimmed  size
{       is to be returned.
{

 FUNCTION [INLINE] clp$trimmed_string_size
   (    str: string ( * )): integer;
```

11.1.15 osp$change_interaction_style

```
{   This request changes the preferred interaction style selected for an
{ interactive job.
{
{       OSP$CHANGE_INTERACTION_STYLE (INTERACTION_STYLE, STATUS)
{
{ INTERACTION_STYLE: (input)  This parameter specifies the selected
{       interaction style.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:  ose$bad_interaction_style
```

```
{

 PROCEDURE [XREF] osp$change_interaction_style
   (    interaction_style: ost$interaction_style;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$interaction_style
*copyc ost$status
?? POP ??
```

11.1.16 osp$compress_file_reference

```
{
{    The purpose of this request is to compress a file_reference to a size less
{ than or equal to the size of the string specified as the
{ compressed_file_reference. This representation of the file reference is not
{ useable as a file_reference but only as a string that is to be displayed
{ later and in which storage is limited. (ie. Placing file_references in a
{ status variable). This representation is not directly useable for displaying
{ the file_reference, but osp$expand_file_reference must be called first. The
{ result of calling osp$expand_file_reference will be to reverse some of the
{ encryption used by osp$compress_file_reference, but it may not reverse all of
{ it. Therefore, the product of osp$compress_file_reference and/or
{ osp$expand_file_reference can never be used for anything other than
{ displaying a representation of the file_reference. The call to
{ osp$expand_file_reference must be executed within the same job as
{ osp$compress_file_reference was executed to ensure correctness. The
{ encryption algorithm used by osp$compress_file_reference may change and
{ therefore can not be relied upon.
{
{       OSP$COMPRESS_FILE_REFERENCE (FILE_REFERENCE, COMPRESSED_FILE_REFERENCE,
{            COMPRESSED_FILE_REFERENCE_SIZE, STATUS)
{
{ FILE_REFERENCE: (input)  This parameter specifies the file_reference that is
{       to be compressed.
{
{ COMPRESSED_FILE_REFERENCE:  (input/output) This parameter specifies the
{       string into which the compressed representation is to be placed. The
{       size of the specified string determines the maximum size for the
{       compressed representation.
{
{ COMPRESSED_FILE_REFERENCE_SIZE: (output)  This parameter specifies the actual
{       size of the compressed representation. This value may be smaller than
{       the size of the string specified as the COMPRESSED_FILE_REFERENCE, but
{       it can never be larger.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:  none.
{

 PROCEDURE [XREF] osp$compress_file_reference
   (    file_reference: fst$file_reference;
    VAR compressed_file_reference: fst$file_reference;
    VAR compressed_file_reference_size: fst$path_size;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc fst$file_reference
```

```
*copyc fst$path_size
*copyc ost$status
?? POP ??
```

11.1.17 osp$expand_file_reference

```
{
{    The purpose of this request is to reverse some of the encryption used by
{ osp$compress_file_reference, but it may not reverse all of it. The product
{ of osp$expand_file_reference can never be used for anything other than
{ displaying a representation of the file_reference. The call to
{ osp$expand_file_reference must be executed within the same job as
{ osp$compress_file_reference was executed to ensure correctness. The
{ encryption algorithm used by osp$compress_file_reference may change and
{ therefore can not be relied upon.
{
{       OSP$EXPAND_FILE_REFERENCE (FILE_REFERENCE, EXPANDED_FILE_REFERENCE,
{             EXPANDED_FILE_REFERENCE_SIZE, STATUS)
{
{ FILE_REFERENCE: (input)  This parameter specifies the file_reference that is
{       to be expanded. .
{
{ EXPANDED_FILE_REFERENCE:  (input/output) This parameter specifies the string
{       into which the expanded representation is to be placed. If the size of
{       the specified string is not large enough to contain the
{       expanded_file_reference, then the reversing of the encryption will not
{       take place.
{
{ EXPANDED_FILE_REFERENCE_SIZE: (output)  This parameter specifies the actual
{       size of the expanded representation.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:  none.
{

 PROCEDURE [XREF] osp$expand_file_reference
   (    file_reference: fst$file_reference;
    VAR expanded_file_reference: fst$file_reference;
    VAR expanded_file_reference_size: fst$path_size;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc fst$file_reference
*copyc fst$path_size
*copyc ost$status
?? POP ??
```

11.1.18 osp$get_interaction_style

```
{
{   This request returns the preferred interaction style selected for an
{ interactive job. Applications that can operate in more than one inter-
{ action style should use this request to determine the user's preference.
{
{       OSP$GET_INTERACTION_STYLE (INTERACTION_STYLE, STATUS)
{
{ INTERACTION_STYLE: (output)  This parameter specifies the prefered
{       interaction style.
```

```
{
{ STATUS: (output)  This parameter specifies the request status.
{

 PROCEDURE [XREF] osp$get_interaction_style
   (VAR interaction_style: ost$interaction_style;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$interaction_style
*copyc ost$status
?? POP ??
```

11.2 Miscellaneous Services Type Declarations

11.2.1 clt$command_log_option

```
 TYPE
   clt$command_log_option = (clc$automatically_log, clc$manually_log);
```

11.2.2 clt$command_or_function_scope

```
 TYPE
   clt$command_or_function_scope = (clc$xdcl_command_or_function,
         clc$gate_command_or_function, clc$local_command_or_function);
```

11.2.3 clt$input_procedure

```
 TYPE
   clt$input_procedure = ^procedure (VAR line: ^clt$command_line;
                                     VAR status: ost$status);

*copyc clt$command_line
*copyc ost$status
```

11.2.4 clt$lexical_token

```
 TYPE
   clt$lexical_token = record
     text_index: clt$string_index,
     text_size: clt$string_size,
     descriptor: string (osc$max_name_size),
     case kind: clt$lexical_token_kind of
     = clc$unknown_token .. clc$string_token =
       str: ost$string,
       str_complete: boolean,
     = clc$unsigned_integer_token, clc$signed_integer_token =
       int: clt$integer,
     = clc$unsigned_real_token, clc$signed_real_token =
       rnum: clt$real,
     casend,
   recend;

*copyc clt$integer
```

```
*copyc clt$lexical_token_kind
*copyc clt$real
*copyc clt$string_index
*copyc clt$string_size
*copyc ost$name
*copyc ost$string
```

11.2.5 clt$lexical_token_kind

```
TYPE
  clt$lexical_token_kind = (clc$unknown_token, clc$end_of_line_token,
        clc$space_token, clc$comment_token, clc$semicolon_token,
        clc$colon_token, clc$cybil_assign_token, clc$left_parenthesis_token,
        clc$right_parenthesis_token, clc$comma_token, clc$ellipsis_token,
        clc$dot_token, clc$query_token, clc$greater_than_token,
        clc$greater_equal_token, clc$less_than_token, clc$less_equal_token,
        clc$equal_token, clc$not_equal_token, clc$concatenate_token,
        clc$exponentiate_token, clc$multiply_token, clc$divide_token,
        clc$add_token, clc$subtract_token, clc$left_bracket_token,
        clc$reverse_slant_token, clc$right_bracket_token,
        clc$circumflex_token, clc$grave_accent_token, clc$left_brace_token,
        clc$vertical_bar_token, clc$right_brace_token, clc$tilde_token,
        clc$number_sign_token, clc$dollar_sign_token,
        clc$commercial_at_token, clc$underscore_token, clc$name_token,
        clc$cybil_name_token, clc$special_cybil_name_token,
        clc$simple_name_token, clc$cobol_name_token, clc$string_token,
        clc$unsigned_integer_token, clc$signed_integer_token,
        clc$unsigned_real_token, clc$signed_real_token);

CONST
  clc$assign_token = clc$equal_token;
```

11.2.6 clt$named_entry_availability

```
TYPE
  clt$named_entry_availability = (clc$normal_usage_entry, clc$hidden_entry,
        clc$advanced_usage_entry);

CONST
  clc$advertised_entry = clc$normal_usage_entry;
```

11.2.7 clt$source

```
TYPE
  clt$source = record
    case kind: clt$source_kind of
    = clc$system_source =
      ,
    = clc$utility_source =
      utility_name: clt$utility_name,
    = clc$catalog_source, clc$library_source =
      path_name: fst$path,
    casend,
  recend;

*copyc clt$source_kind
*copyc clt$utility_name
```

```
*copyc fst$path
```

11.2.8 clt$source_kind

```
TYPE
  clt$source_kind = (clc$system_source, clc$utility_source,
        clc$catalog_source, clc$library_source);
```

11.2.9 clt$task_name

```
TYPE
  clt$task_name = ost$name;

*copyc ost$name
```

11.2.10 clt$task_name_reference

```
TYPE
  clt$task_name_reference = string ( * <= osc$max_name_size);

*copyc ost$name
```

11.2.11 clt$token_evaluation_option

```
TYPE
  clt$token_evaluation_option = (clc$ignore_spaces_before_token,
        clc$comment_is_token, clc$classify_name_token,
        clc$cobol_name_is_token, clc$special_cybil_name_is_token,
        clc$international_char_is_token, clc$special_char_is_token);
```

11.2.12 lt$token_evaluation_options

```
TYPE
  clt$token_evaluation_options = set of clt$token_evaluation_option;

*copyc clt$token_evaluation_option
```

11.2.13 clt$type_information

```
TYPE
  clt$type_information = record
    case kind: clt$type_kind of
    = clc$application_type =
      balance_brackets: boolean,
    = clc$array_type =
      array_element_type_information: ^clt$type_information,
      case array_bounds_defined: boolean of
      = TRUE =
        bounds: clt$array_bounds,
      = FALSE =
        ,
      casend,
    = clc$boolean_type =
      ,
```

```
= clc$cobol_name_type =
  ,
= clc$command_reference_type =
  ,
= clc$data_name_type =
  ,
= clc$date_time_type =
  date_and_or_time: clt$date_and_or_time,
  tenses: clt$date_time_tenses,
= clc$entry_point_reference_type =
  ,
= clc$file_type =
  ,
= clc$integer_type =
  min_integer_value: integer,
  max_integer_value: integer,
  default_radix: 2 .. 16,
= clc$keyword_type =
  keyword_specifications: ^clt$keyword_specifications,
= clc$list_type =
  list_element_type_information: ^clt$type_information,
  min_list_size: clt$list_size,
  max_list_size: clt$list_size,
  list_rest: boolean,
= clc$lock_type =
  ,
= clc$name_type =
  min_name_size: ost$name_size,
  max_name_size: ost$name_size,
= clc$network_title_type =
  ,
= clc$program_name_type =
  ,
= clc$range_type =
  range_element_type_information: ^clt$type_information,
= clc$real_type =
  min_real_value: longreal,
  max_real_value: longreal,
= clc$record_type =
  fields_information: ^array [1 .. * ] of clt$field_information,
= clc$scu_line_identifier_type =
  ,
= clc$statistic_code_type =
  ,
= clc$status_type =
  ,
= clc$status_code_type =
  ,
= clc$string_type =
  min_string_size: clt$string_size,
  max_string_size: clt$string_size,
  literal: boolean,
= clc$string_pattern_type =
  ,
= clc$time_increment_type =
  ,
= clc$time_zone_type =
  ,
= clc$type_specification_type =
  ,
```

```
      = clc$union_type =
        members_information: ^array [1 .. * ] of clt$type_information,
      casend,
    recend;

*copyc clt$array_bounds
*copyc clt$date_and_or_time
*copyc clt$date_time_tenses
*copyc clt$keyword_specifications
*copyc clt$field_information
*copyc clt$list_size
*copyc clt$string_size
*copyc clt$type_kind
*copyc ost$name

 TYPE
   clt$field_information = record
     name: clt$field_name,
     requirement: clt$field_requirement,
     type_information: clt$type_information,
   recend;

*copyc clt$field_name
*copyc clt$field_requirement
*copyc clt$type_information
```

11.2.14 ost$interaction_style

```
 TYPE
   ost$interaction_style = (osc$line_interaction, osc$screen_interaction,
         osc$desktop_interaction);
```

11.2.15 ost$string

```
 CONST
   osc$max_string_size = 256;

 TYPE
   ost$string_size = 0 .. osc$max_string_size;

 TYPE
   ost$string_index = 1 .. osc$max_string_size + 1;

 TYPE
   ost$string = record
     size: ost$string_size,
     value: string (osc$max_string_size),
   recend;
```

11.2.16 pmt$task_id

```
 TYPE
   pmt$task_id = 0 .. pmc$max_task_id;

 CONST
   pmc$max_task_id = 0ffffffff(16);
```

11.2.17 pmt$task_status

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

*copyc OST$STATUS
```

12 Conversions from Strings

12.1 Conversions from Strings Interfaces

12.1.1 clp$convert_string_to_date_time

Detailed information on the contents of "date time form strings" can be found in the *Date and Time Formats* section of the SCL Command Interface ERS.

```
{
{   This request interprets the string representation of a date and/or time to
{ produce a formal representation of the date/time.
{
{       CLP$CONVERT_STRING_TO_DATE_TIME (STR, FORMAT, DATE_TIME, STATUS)
{
{ STR: (input)  This parameter specifies the string to be interpreted.
{
{ FORMAT: (input)  This  parameter specifies the "date time form string" to be
{       used to guide the interpretation. If the format is null or all blanks
{       an  attempt  is  made  to  match  the  input string against one of the
{       standard "date time form strings".
{
{ DATE_TIME: (output)  This parameter specifies the interpreted date/time.
{
{ STATUS: (output)  This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$convert_string_to_date_time
   (    str: string ( * );
        format: clt$date_time_form_string;
    VAR date_time: clt$date_time;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$date_time
*copyc clt$date_time_form_string
*copyc ost$status
?? POP ??
```

12.1.2 clp$convert_string_to_file_ref

```
{
{   The purpose of this request is to convert a string to a file reference.
{ The string is assumed to contain a file expression. The returned parsed file
{ reference provides for accessing the entire path of the file reference or its
{ components.
{
{       CLP$CONVERT_STRING_TO_FILE_REF (STR, PARSED_FILE_REFERENCE, STATUS)
{
{ STR: (input)  This parameter specifies the string to be converted.
{
{ PARSED_FILE_REFERENCE: (output)  This parameter specifies the file reference.
{
{ STATUS: (output)  This parameter specifies the request status.
```

```
{

 PROCEDURE [XREF] clp$convert_string_to_file_ref
   (    str: string ( * );
    VAR parsed_file_reference: fst$parsed_file_reference;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc fst$parsed_file_reference
*copyc ost$status
?? POP ??
```

12.1.3 clp$convert_string_to_integer

```
{
{   The purpose  of this request is to convert the string representation of an
{ integer to an integer. The string representation may contain a leading sign
{ (+ or -) and/or a trailing radix enclosed in parentheses.
{
{       CLP$CONVERT_STRING_TO_INTEGER (STR, INT, STATUS)
{
{ STR: (input) This parameter specifies the string to be converted.
{
{ INT: (output)  This  parameter  specifies  the converted integer value along
{       with the radix in which the integer was represented.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: clc$min_ecc_lexical .. clc$max_ecc_lexical
{       IDENTIFIER: 'CL'
{

 PROCEDURE [XREF] clp$convert_string_to_integer ALIAS 'clpcs2i'
   (    str: string ( * );
    VAR int: clt$integer;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_lexical
*copyc clt$integer
*copyc ost$status
?? POP ??
```

12.1.4 clp$convert_string_to_name

```
{
{   The purpose  of  this  request  is  to  convert  a  string to a name. The
{ conversion invloves  padding  the  name  with  spaces  (on  the  right)  and
{ transforming lower case letters into their upper case counterparts.
{
{       CLP$CONVERT_STRING_TO_NAME (STR, NAME, STATUS)
{
{ STR: (input) This parameter specifies the string to be converted.
{
{ NAME: (output) This parameter specifies the converted name and its length.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: clc$min_ecc_lexical .. clc$max_ecc_lexical
{       IDENTIFIER: 'CL'
```

```
{

 PROCEDURE [XREF] clp$convert_string_to_name ALIAS 'clpcs2n'
   (    str: string ( * );
    VAR name: clt$name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_lexical
*copyc clt$name
*copyc ost$status
?? POP ??
```

12.1.5 clp$convert_string_to_real

```
{
{   This request converts the string representation of a real number to a real
{ number. The string representation may contain leading sign (+ or -).
{
{       CLP$CONVERT_STRING_TO_REAL (STR, REAL_NUMBER, STATUS)
{
{ STR: (input) This parameter specifies the string to be converted.
{
{ REAL_NUMBER: (output) This parameter specifies the resulting real number.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$convert_string_to_real
   (    str: string ( * );
    VAR real_number: clt$real;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$real
*copyc ost$status
?? POP ??
```

12.2 Conversions from Strings Type Declarations

12.2.1 fst$parsed_file_reference

```
{
{ A FST$PARSED_FILE_REFERENCE provides a breakdown of a file reference. The
{ path field is a container for the maximum-sized file reference in the form of
{ a string.
{
{ The remaining fields define a substring within the path field for particular
{ path elements. Depending upon the context in which this type is used, not
{ all of the path elements may be defined. A missing path element will be
{ identified by a null string, i.e. the index will be set to one (1) and the
{ size will be set to zero (0).
{
{ The information provided by these fields is best described by example:
{
```

```
{ Assume the path is:  :NVE.AJL.BAM.SOURCE_LIBRARY.$HIGH.$ASIS
{
{ complete_path provides ->        :NVE.AJL.BAM.SOURCE_LIBRARY.$HIGH.$ASIS
{ cycle_path provides ->           :NVE.AJL.BAM.SOURCE_LIBRARY.$HIGH
{ file_path provides ->            :NVE.AJL.BAM.SOURCE_LIBRARY
{ catalog_path provides ->         :NVE.AJL.BAM
{ first_name provides ->           NVE
{ last_name provides ->            SOURCE_LIBRARY
{ cycle_reference provides ->      $HIGH
{ open_position provides ->        $ASIS
{ number_of_path_elements ->       4
{
{ Assume the path is:  $LOCAL.JUNK.1.$ASIS
{
{ complete_path provides ->        :$LOCAL.JUNK.1.$ASIS
{ cycle_path provides ->           :$LOCAL.JUNK.1
{ file_path provides ->            :$LOCAL.JUNK
{ catalog_path provides ->         :$LOCAL
{ first_name provides ->           $LOCAL
{ last_name provides ->            JUNK
{ cycle_reference provides ->      1
{ open_position provides ->        $ASIS
{ number_of_path_elements ->       2
{
{ Assume the path is:  :$SOURCE.$UP.OBJECT_LIBRARY
{
{ complete_path provides ->        :$SOURCE.$UP.OBJECT_LIBRARY
{ cycle_path provides ->           :$SOURCE.$UP.OBJECT_LIBRARY
{ file_path provides ->            :$SOURCE.$UP.OBJECT_LIBRARY
{ catalog_path provides ->         :$SOURCE.$UP
{ first_name provides ->           $SOURCE
{ last_name provides ->            OBJECT_LIBRARY
{ cycle_reference provides ->      null string
{ open_position provides ->        null string
{ number_of_path_elements ->       3
{

 TYPE
   fst$parsed_file_reference = record
     path: fst$path,
     catalog_path_size: fst$path_size,
     complete_path_size: fst$path_size,
     cycle_path_size: fst$path_size,
     cycle_reference: fst$path_element_substring,
     file_path_size: fst$path_size,
     first_name: fst$path_element_substring,
     last_name: fst$path_element_substring,
     number_of_path_elements: fst$number_of_path_elements,
     open_position: fst$path_element_substring,
   recend;

*copyc fst$number_of_path_elements
*copyc fst$path
*copyc fst$path_size
*copyc fst$path_element_substring
```

13 Conversions to Strings

13.1 Conversions to Strings Interfaces

13.1.1 clp$convert_data_to_string

```
{
{   The purpose of this request is to convert a clt$data_value to its string
{ representation. The representation may take on a number of forms determined
{ by the REPRESENTATION_OPTION parameter.
{
{   This representation is returned in a sequence. The first data item in the
{ sequence is a clt$data_representation_count, i.e. the number of strings
{ used in the representation. The remaining data is a series of
{ clt$string_size, clt$string_value pairs with the length of each
{ clt$string_value determined by the corresponding clt$string_size.
{
{       CLP$CONVERT_DATA_TO_STRING (VALUE, REPRESENTATION_OPTION, MAX_STRING,
{         WORK_AREA, DATA_REPRESENTATION, STATUS)
{
{ VALUE: (input)  This parameter specifies the data value to be converted.
{
{ REPRESENTATION_OPTION: (input)  This parameter specifies the form of the
{       data value's representation as determined by the following options:
{
{       CLC$DATA_ELEM_REPRESENTATION:  each element of the data value is
{             converted to a separate string in the representation. Status
{             values are formatted as messages and may therefore occupy more
{             than one string. The built-in record types command_reference,
{             date_time, entry_point_reference, scu_line_identifier,
{             time_increment and time_zone are considered to be elements for
{             purposes of this option.
{
{       CLC$DATA_STRUCT_REPRESENTATION:  each element of the data value is
{             converted to a separate string in the representation. The
{             generic data type of the value (and components of the value, if
{             applicable) are included in the representation as "comments"
{             prefixing the corresponding value or value component. Items that
{             are components of data structures are prefixed with identifying
{             "labels", e.g. field names for record elements, subscripts for
{             array elements, etc.
{
{       CLC$DATA_SOURCE_REPRESENTATION:  the data value is represented in the
{             form of an SCL "expression" that if evaluated with an appropriate
{             type specification would yield exactly the same data value. All
{             strings in the representation, except the last, will end with an
{             ellipsis ("..").
{
{ MAX_STRING: (input)  This parameter specifies the maximum length of strings
{       in the representation of the value.
{
{ WORK_AREA: (input, output)  This parameter specifies an area of storage that
{       will be used to construct the value's representation. The current
{       position of this sequence pointer is updated to reflect the amount of
{       storage used by the request. The value's respresentation occupies the
{       used part of this sequence.
```

```
{
{ DATA_REPRESENTATION: (output)  This parameter specifies the data value's
{       representation as described above.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$work_area_overflow
{                   cle$bad_data_rep_option
{                   cle$bad_data_value
{

 PROCEDURE [XREF] clp$convert_data_to_string
   (    value: ^clt$data_value;
        representation_option: clt$data_representation_option;
        max_string: clt$string_size;
    VAR work_area {input, output} : ^clt$work_area;
    VAR data_representation: ^clt$data_representation;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$bad_data_rep_option
*copyc cle$bad_data_value
*copyc cle$work_area_overflow
*copyc clt$data_representation
*copyc clt$data_representation_option
*copyc clt$data_value
*copyc clt$string_size
*copyc clt$work_area
*copyc ost$status
?? POP ??
```

13.1.2 clp$convert_date_time_to_string

Detailed information on the contents of “date time form strings” can be found in the *Date and Time Formats* section of the SCL Command Interface ERS.

```
{
{   This request produces the string representation  of  a  date  and/or  time
{ and/or day of the week.
{
{       CLP$CONVERT_DATE_TIME_TO_STRING (DATE_TIME, FORMAT, STR, STATUS)
{
{ DATE_TIME: (input) This parameter specifies the date/time to be represented.
{
{ FORMAT: (input) This parameter specifies the "date time form string"  to  be
{       used to guide the construction of the representation.
{
{ STR: (output) This parameter specifies the result string.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$convert_date_time_to_string
   (    date_time: clt$date_time;
        format: clt$date_time_form_string;
    VAR str: ost$string;
    VAR status: ost$status);
```

```
?? PUSH (LISTEXT := ON) ??
*copyc clt$date_time
*copyc clt$date_time_form_string
*copyc ost$status
*copyc ost$string
?? POP ??
```

13.1.3 clp$convert_integer_to_string

```
{
{   The purpose  of  this  request  is  to  convert  an  integer to its string
{ representation in a specified radix. The result is left  justified  in  the
{ returned  string.  If  the  integer is negative, the first character of the
{ result is a minus sign (-). If the specified radix is greater than ten  and
{ the  leftmost  digit of the result is greater than nine, then a leading zero
{ digit is added to the result.
{
{       CLP$CONVERT_INTEGER_TO_STRING (INT, RADIX, INCLUDE_RADIX_SPECIFIER,
{         STR, STATUS)
{
{ INT: (input) This parameter specifies the integer to be converted.
{
{ RADIX:  (input)  This  parameter  specifies the radix in which the integer's
{       value is to be represented.
{
{ INCLUDE_RADIX_SPECIFIER:  (input)  This  parameter  specifies  whether   the
{       representation  of the radix is to be included in the resulting string
{       -- e.g. (16) for a number with a radix of 16.
{
{ STR: (output) This parameter specifies  the  string  representation  of  the
{       integer.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{       IDENTIFIER: 'CL'
{

 PROCEDURE [XREF] clp$convert_integer_to_string ALIAS 'clpci2s'
   (    int: integer;
        radix: 2 .. 16;
        include_radix_specifier: boolean;
    VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$string
?? POP ??
```

13.1.4 clp$convert_integer_to_rjstring

```
{
{   The purpose  of  this  request  is  to  convert  an  integer to its string
{ representation in a specified radix. The result is right justified  in  the
{ returned  string.  If the integer is negative, a minus sign (-) is included
{ in the result either just to the left of the converted integer if  the  fill
{ character is a space, or as the leftmost character of the result string. If
{ the specified radix is greater than ten and the leftmost digit of the result
```

```
{ is  greater  than  nine, then a leading zero digit is added to the result if
{ the result string is long enough to hold it.
{
{       CLP$CONVERT_INTEGER_TO_RJSTRING (INT, RADIX, INCLUDE_RADIX_SPECIFIER,
{         FILL_CHARACTER, STR, STATUS)
{
{ INT: (input) This parameter specifies the integer to be converted.
{
{ RADIX:  (input)  This  parameter  specifies the radix in which the integer's
{       value is to be represented.
{
{ INCLUDE_RADIX_SPECIFIER:  (input)  This  parameter  specifies  whether   the
{       representation  of the radix is to be included in the resulting string
{       -- e.g. (16) for a number with a radix of 16.
{
{ FILL_CHARACTER: (input) This parameter specifies the character used to  fill
{       unused positions in the returned string.
{
{ STR: (output)  This  parameter  specifies  the  string representation of the
{       integer.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: cle$string_too_short
{       IDENTIFIER: 'CL'
{

 PROCEDURE [XREF] clp$convert_integer_to_rjstring ALIAS 'clpcirs'
   (    int: integer;
        radix: 2 .. 16;
        include_radix_specifier: boolean;
        fill_character: char;
    VAR str: string ( * );
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cle$ecc_expression_result
*copyc ost$status
?? POP ??
```

13.1.5 clp$convert_real_to_string

```
{
{   This request converts a real number to  its  string  representation.  The
{ result  is  left  justified  in  the returned string. If the real number is
{ negative, the first character of the result is a minus  sign  (-).  If  the
{ absolute value of the real number is less than or equal to 10**-6 or greater
{ than 10**9, then an exponent is included in the result.
{
{       CLP$CONVERT_REAL_TO_STRING (REAL_NUMBER, NUMBER_OF_DIGITS, STR,
{         STATUS)
{
{ REAL_NUMBER:  (input)  This  parameter  specifies  the  real  number  to  be
{       converted.
{
{ NUMBER_OF_DIGITS:  (input)  This  parameter  specifies the maximum number of
{       digits to include in the result. There may be fewer digits than  this
{       since zero  digits to the right of the decimal point are not included.
{
{ STR: (output) This parameter specifies the string representation of the real
```

```
{       number.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] clp$convert_real_to_string
   (    real_number: longreal;
        number_of_digits: clt$real_number_digit_count;
    VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$real_number_digit_count
*copyc ost$status
*copyc ost$string
?? POP ??
```

13.1.6 clp$data_representation_text

```
{
{   This function is intended for use following a call to
{ CLP$CONVERT_DATA_TO_STRING. Given a pointer to the "data representation"
{ produced by that request it returns a pointer to the first "line" of that
{ representation. It is intended as a convenience; for use only in those cases
{ where the representation consists of exactly one "line".
{
{       CLP$DATA_REPRESENTATION_TEXT (REPRESENTATION):  TEXT
{
{ REPRESENTATION: (input)  This parameter specifies the data representation
{       produced by CLP$CONVERT_DATA_TO_STRING.
{
{ TEXT: (result)  The function's result specifies a pointer to the first "line"
{       within the representation.
{

 FUNCTION [INLINE] clp$data_representation_text
    (    representation: ^clt$data_representation): ^clt$string_value;

?? PUSH (LISTEXT := ON) ??
*copyc clt$data_representation
*copyc clt$string_value
?? POP ??
```

13.2 Conversions to Strings Type Declarations

13.2.1 clt$data_representation

```
 TYPE
   clt$data_representation = SEQ ( * );

*copyc clt$string_size
*copyc clt$string_value
*copyc clt$data_representation_count
```

13.2.2 clt$data_representation_count

```
TYPE
  clt$data_representation_count = 0 .. 7fffffff(16);
```

13.2.3 clt$data_representation_option

```
TYPE
  clt$data_representation_option = (clc$data_elem_representation,
        clc$data_struct_representation, clc$data_source_representation);
```

14 Status and Message Handling

NOS/VE and its facilities use the status record to convey the completion status of requests. To insure a consistent and meaningful response is presented to the command level user, the system message generator which has the status record as its primary input, is used. The condition code in the status record is used to find the corresponding message template. The message template and the text field of the status record are combined into a standard NOS/VE formatted message.

14.1 Severity Levels

The severity level associated with an exception (status) condition is used in various contexts to determine how the condition should affect subsequent processing. For example, the SCL interpreter will abort the processing of commands in a "block" if one of the commands terminates with a status whose severity is "error" or above. The levels are described below in order of increasing severity.

INFORMATIVE conditions designate items of general interest and will not cause any problems with subsequent operations.

WARNING conditions designate items of general interest which may cause incorrect or incomplete processing of subsequent operations.

ERROR conditions designate that a command or request was not completed correctly because of something which the user would normally be able to correct. Subsequent processing may be provided in order to discover additional problems.

FATAL conditions designate that a command or request was not completed correctly becuase of something which the user would not normally be able to correct. Subsequent processing may be provided in order to discover additional problems.

CATASTROPHIC conditions designate that a command or request was not completed correctly because of something which the user would not normally be able to correct. No further processing was possible by the command or request.

The above are the severity levels for status conditions. Another set of severity levels are provided for "diagnostics" produced by programs like compilers. The diagnostic severity levels include those described above plus the following.

NON_STANDARD diagnostics designate items that do not conform to applicable standards but are supported.

DEPENDENT diagnostics designate items that are machine dependent.

Both of these diagnostic severity levels are treated as INFORMATIVE when used as a status severity level.

14.2 Message Template Text

A message template is a string of text characters and control sequences. Each text character in the template is copied to the output text when a message is formatted.

The "+" character is used to introduce a control sequence within a template. This control character and the characters immediately following it designate actions the message formatter is to perform with respect to the output text. In the following descriptions of the control sequences, the notation [n] is used to indicate that n is optional. The control sequences are:

+P[n] Edit the nth message parameter from the text field of the status record into the output text (n is in the range 1 to 128). If n is omitted, the next message parameter from the text field is used.

+F[n] Use the nth message parameter from the text field of the status record, assume it designates a file, expand it to a complete file reference and edit it into the output text (n is in the range 1 to 128). If n is omitted, the next message parameter from the text field is used.

+N[n] Start a new line indented by n spaces in the output text. If n is omitted, zero is used.

+E[n] If the current line of output text must be continued onto another line due to output file or device restrictions, start the new line at the point where the most recent +E was encountered and indent it by n spaces. If n is omitted, zero is used. This control sequence is known as a "soft" or "conditional" new line.

+X[n] Place n spaces in the output text. If n is omitted, one is used. This control sequence must be used whenever two or more contiguous spaces are desired in the resulting message since contiguous spaces in the message template and/or parameters are compressed into a single space.

+K	This control sequence is used in pairs to bracket message text that should be kept together on a line if possible
+S	Edit the severity level into the output text.
+C	Edit the condition field from the status record into the output text.
+I	Edit the “product identifier” portion of the condition field from the status record into the output text.
+T	Copy the entire text field from the status record to the output text.
+H[n]	Place spaces into the output text until column n of the current line is reached, i.e. horizontal tab to column n. If n is less than or equal to the current column number, do nothing. If n is greater than the maximum line size for the formatting request, start a new line with no indentation. If n is omitted, use the next available default tab that is greater than the current column number. The default tabs are 1, 9, 17, 25, and so on.
+R	Repeat the remainder of the template until all message parameters in the text field of the status record have been edited into the output text. If at the point where this control sequence is encountered, there are no more message parameters, the remainder of the template is not processed. If no +P or +F control sequences, or any +P's or +F's with a specified message parameter number, appear between the +R and the end of the template, that part of the template is processed only once. This control sequence may be used only once in a template.
++	The combination of two consecutive control characters represents the control character itself. A plus is placed into the output text.
+-	The combination of a plus followed by a minus is treated as a “do nothing.” It allows substituted text to be followed by template text with no intervening spaces.

In the +P[n] and +F[n] control sequences the n serves as a place holder, i.e. a subsequent +P or +F would reference the nth+1 message parameter from the text field of the status record.

Alphabetic control characters may appear in the template in either upper or lower case.

If the character following the control character is not one of those described above, the message formatter will not take any action but outputs the characters verbatim.

14.3 Message Parameter Text

Parameters for insertion into formatted messages are obtained from the text field of the status record. The first character of the text field serves as the “status parameter delimiter,” i.e. it is used to separate message parameters from one another within the text field. The character used for this purpose by the OSP$SET_STATUS_ABNORMAL interface is designated in CYBIL by the OSC$STATUS_PARAMETER_DELIMITER constant identifier which, in turn, designates the ASCII unit separator character (US).

14.4 Formatted Message Levels and Other Conventions

The “message level” selected for the job (brief or full) affects message formatting as described below.

When a message is formatted, a “header” is prefixed to it. If the message level is full or the severity of the condition is greater than or equal to warning, the severity level appears in the message header. If the message level is brief, the condition code is edited into the message header.

Examples of brief mode messages

```
informative message:

      --  No entry point cross references detected.

warning message:
```

```
      --WARNING--  JUNK is not in job command list.

error message:

      --ERROR--  MORK is not a command.

fatal message:

      --FATAL--  Starting procedure not found.

catastrophic message:

      --CATASTROPHIC--  Overflow of Program Segment TEST.
```

Examples of full mode messages

```
informative message:

      --INFORMATIVE LL 104--  No entry point cross
      references detected.

warning message:

      --WARNING CL 755--  JUNK is not in job command list.

error message:

      --ERROR CL 790--  MORK is not a command.

fatal message:

      --FATAL LL 102--  Starting procedure not found.

catastrophic message:

      --CATASTROPHIC LL 380--  Overflow of Program
      Segment TEST.
```

The message level also affects the way in which a path name for a file or catalog is edited into a message. If the message level is brief, and the file or catalog in question is contained within the working catalog for the job, the path relative to the working catalog is placed in the formatted message; otherwsie the complete path name is used.

The code for an exception (status) condition can be thought of as consisting of two parts:

1. a two character “product identifier,” for example 'CL' for the NOS/VE command language,
2. a number in the range 0 to 16777215.

The numbers in the range 0 to 9999 for every possible product identifier are reserved for use by Control Data. The numbers in the range 10000 to 19999 for every possible product identifier are reserved for user developed products. The remainder of each range is reserved for future use.

When a condition code is edited into a message, it always appears in the form:

XX n

where XX is the product identifier and n is the numeric part of the code.

14.5 Natural Language and Message Templates

NOS/VE supports the simultaneous existance of a number of versions of message templates for the same exception (status)

condition. Each version is written in a different natural language (e.g. Danish, Dutch, English, Finnish, Flemish, French, German, Italian, Norwegian, Portugese, Spanish, Swedish, US_English).

All message templates in a given message module are assumed to be written in the same natural language, and the name of that language is specified on the CREATE_MESSAGE_MODULE command (the default natural language is US_English).

14.6 Creating Message Templates

The descriptive information about an exception (status) condition resides in a "message module." This descriptive information consists of the name of the condition, the condition code, its severity and the template for its message.

A message module is built using the CREATE_MESSAGE_MODULE subutility of the CREATE_OBJECT_LIBRARY utility. Within the CREATE_MESSAGE_MODULE subutility, the CREATE_STATUS_MESSAGE subcommand is used to describe an individual exception condition.

For CYBIL programmers, the GENERATE_MESSAGE_TEMPLATES command can be used to "front end" message module creation so that the setting of status records and the production of the corresponding descriptive information is more readily coordinated.

14.6.1 generate_message_template(s) (genmt)

GENERATE_MESSAGE_TEMPLATES (GENMT) is a tool for use by CYBIL programmers to facilitate the creation of message modules. A message module contains descriptive information about status conditions including their names, severity levels and message templates.

This tool is useful for the coordination of setting status records and the construction of the information that describes the corresponding conditions. This is accomplished by supplying as input to this tool, the definitions of the CYBIL constant identifiers for condition codes along with CYBIL comments which are interpreted to represent the severity level and message template of their associated conditions.

The output produced by GENERATE_MESSAGE_TEMPLATES is a file containing commands that when processed via the INCLUDE_FILE command within the CREATE_OBJECT_LIBRARY utility will build the desired message module.

```
generate_message_templates (
      input, i: file = $required
      output, o: file = $required
      error, e: file = $errors
      product_identifier, identifier, pi: name 2..2 = $optional
      status)
```

INPUT, I: This parameter specifies the file containing the CYBIL definitions for the condition codes, their severity levels and their message templates. See the next subsection for information and examples on how to construct this input.

The FILE_CONTENTS attribute of this file must be LEGIBLE_DATA or UNKNOWN and the FILE_PROCESSOR attribute must be CYBIL or UNKNOWN.

OUTPUT, O: This parameter specifies the file to receive the message module creation commands. This file must be included within the CREATE_OBJECT_LIBRARY utility to actually create the message module(s).

If GENERATE_MESSAGE_TEMPLATES creates the output file, its FILE_CONTENTS attribute is set to LEGIBLE_SCL_INCLUDE.

ERROR, E: This parameter specifies the file to which information about errors encountered in the input is written.

Omission causes $ERRORS to be used.

PRODUCT_IDENTIFIER, IDENTIFIER, PI: This parameter can be used to specify the product identifier portion of all status code definitions that are generated. Specifying this parameter overrides any product identifiers that are defined in the input file.

Omission causes the product identifier for a status code to be derived from the information in the input file.

STATUS: See the Error Handling section of the SCL Command Interface ERS.

The format and processing of the input file is as follows:

- Any line beginning with a CYBIL name where the third and fourth characters are c$, c#, e$, or e# is considered a constant to be evaluated. An e$ or e# constant is considered to be the start of a condition code definition.
- A constant may be declared in the following ways:

  constant = integer

  constant = constant

  constant = constant + integer

  constant = (($INTEGER('x')*100(16))+$INTEGER('y'))*1000000(16)

  constant = (($INTEGER('x')*100(16))+$INTEGER('y'))*1000000(16) + integer

  (In the latter two cases x and y specify the first and second characters, respectively, of the product identifier associated with the code being declared.)

  In the case of an e$ or e# constant the condition name and condition code are defined at this point.
- Only one constant declaration per line is recognized; and the constant declaration must be contained on one line.

* If a c$ or c# constant is declared more than once the first definition is used.

- Any consecutive lines beginning with '{' directly following an e$ or e# constant declaration are processed for the error severity level and message text. The lines may contain a corresponding '}' to signify the end of the line, or the '}' may be omitted. An line containing an “empty” CYBIL comment (e.g. {}) is traeted as a completely empty line, i.e. it terminates message text collection.

* The severity level indicator is the first character after '{' of the first line. A character is recognized as a severity indicator if it is the first letter (either upper or lower case) of the name of the severity level. If one is not found error (E) is assumed.

- An ellipsis may be optionally included at the end of a line of message text if the text is continued onto the next line.

* The message text is truncated if it exceeds the maximum size allowed.

- A condition definition will generate a CREATE_STATUS_MESSAGE command written to the output file.
- Any line beginning with 'MODULE' OR 'MODEND' will generate a CREATE_MESSAGE_MODULE or QUIT command, respectively, written to the output file.
- All lines other than those mentioned above and excess characters of processed lines are ignored. For example, the following line would be ignored:

  ignore_constant =1;

  and in the following line:

  xxe$error = 10; {E message text

  the characters starting with ';' through to the end of line are ignored.

Error messages are issued for all errors that may be found while processing a line. At such points the c$ or c# constant declaration or complete condition definition is ignored except for those situations noted above with '*'.

As an example of how SOURCE_CODE_UTILITY (SCU) “common” decks could be used to organize condition definitions, assume the following data is on a file called ECC_DECKS:

```
*DECK CLE$EXCEPTION_CONDITION_CODES, EXPAND=FALSE
?? NEWTITLE := 'cle$exception_condition_codes : CL 1..9999' ??
*copyc clc$ecc_range
*copyc cle$ecc_miscellaneous

{*copyc other CL error code decks here.}
?? OLDTITLE ??
*DECK CLC$ECC_RANGE, EXPAND=FALSE

 CONST
   clc$min_ecc = (($INTEGER ('C') * 100(16)) + $INTEGER ('L')) * 1000000(16),
   clc$max_ecc = clc$min_ecc + 9999;
```

```
*DECK CLE$ECC_MISCELLANEOUS, EXPAND=FALSE
*copyc clc$ecc_range
?? NEWTITLE := 'cle$ecc_miscellaneous : CL 1..99', EJECT ??
?? FMT (FORMAT := OFF) ??

 CONST
   clc$min_ecc_scl                = clc$min_ecc + 0,

   cle$incompatible_params_given  = clc$min_ecc_scl + 40,
   {E Parameters given are incompatible with value of parameter +P.}

   cle$none_must_be_used_alone    = clc$min_ecc_scl + 50,
   {E NONE must be used alone for parameter +P.}

   cle$unexpected_call_to         = clc$min_ecc_scl + 90,
   {F Unexpected call to +P.}

   clc$max_ecc_scl                = clc$min_ecc_scl + 99;

?? FMT (FORMAT := ON) ??
?? OLDTITLE ??
*DECK CLM$MESSAGE_MODULE, EXPAND=TRUE
MODULE clm$message_module;
*copy cle$exception_condition_codes
MODEND clm$message_module;
```

Then the following commands could be used to produce a message module:

```
SOURCE_CODE_UTILITY
  create_deck ddi=true s=ecc_decks
  expand_deck d=clm$message_module c=in
QUIT
generate_message_templates i=in o=out
CREATE_OBJECT_LIBRARY
  add_modules my_object
  include_file out
  generate_library my_object.$next
QUIT
```

The output from GENERATE_MESSAGE_TEMPLATES (file OUT) would be similar to the following:

```
CREATE_MESSAGE_MODULE clm$message_module
 create_status_message name=cle$incompatible_params_given, ..
       identifier='CL', code=40, severity=error, ..
       collect_template_until='**'..
Parameters given are incompatible with value of parameter +P.
**
 create_status_message name=cle$none_must_be_used_alone, ..
       identifier='CL', code=50, severity=error, ..
       collect_template_until='**'..
NONE must be used alone for parameter +P.
**
 create_status_message name=cle$unexpected_call_to, ..
       identifier='CL', code=90, severity=error, ..
       collect_template_until='**'..
Unexpected call to +P.
**
END_MESSAGE_MODULE
```

14.7 Status and Message Interfaces

Several procedure interfaces are provided at the program interface level which assist in constructing a status record and generating a message corresponding to a status record. Interfaces are also provided to set and interrogate various items that affect the search for and presentation of status messages. These interfaces are described in the following subsections.

14.7.1 osp$set_status_abnormal

```
{
{   The purpose of this request is to set  a  status  record  to  represent  a
{ message (usually describing some abnormal condition).
{
{   Additional    text    may    be    appended    to    the    status    via    the
{ osp$append_status_parameter,                        osp$append_status_integer,
{ osp$append_status_file, etc. interfaces.
{
{   These  requests  provide  a basic capability for editing status parameters
{ into the status record's text field (e.g. the removal  of  trailing  spaces
{ from the status parameters).
{
{   The  condition  code for the status may be specified as a single number or
{ via its "component parts", i.e. the two character "product identifier"  and
{ the  "condition  number". If the value given for the condition parameter is
{ within the bounds of a ost$status_condition_number, the identifier parameter
{ is  combined  with  it  to  form  the  condition  code in the status record;
{ otherwise the  condition  parameter  is  assumed  to  represent  the  entire
{ condition code and the identifier parameter is ignored.
{
{       OSP$SET_STATUS_ABNORMAL (IDENTIFIER, CONDITION, TEXT, STATUS)
{
{ IDENTIFIER:  (input) This parameter specifies the product identifier for the
{       condition.
{
{ CONDITION: (input) This parameter specifies the condition code (or condition
{       number) for the condition (see above).
{
{ TEXT: (input) This parameter specifies the first status parameter. The text
{       field's size is initially set to zero (no text). If this parameter is
{       not     the     null     string     (string     length     not     zero)
{       osp$append_status_parameter is called to add the parameter to the text
{       field  with  a  delimiter of osc$status_parameter_delimiter; otherwise
{       the text field is left empty.
{
{ STATUS: (output) This paraeter specifies the status record to be set.
{

 PROCEDURE [XREF] osp$set_status_abnormal ALIAS 'ospssa'
   (    identifier: ost$status_identifier;
        condition: ost$status_condition_code;
        text: string ( * <= osc$max_string_size);
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$status_identifier
*copyc ost$string
?? POP ??
```

14.7.2 osp$append_status_parameter

```
{
{   The purpose of this request is to append a status parameter to the text of
{ a  status  record.  This  request  assumes  that the status record has been
{ initialized by a call to osp$set_status_abnormal.
{
{       OSP$APPEND_STATUS_PARAMETER (DELIMITER, TEXT, STATUS)
{
{ DELIMITER: (input) This parameter specifies the character that will  precede
{       the  text  supplied  by  this  request in the text field of the status
{       being formed. Depending on its value, this character may cause one of
{       two  effects  when  the  status  is formatted into a message. If this
{       character is the same as the first character in the text  field,  then
{       the  new  text  will  be  treated as a separate status parameter. If,
{       however, this character is different from the first character  in  the
{       text  field,  then  this character and the new text will be treated as
{       part of the preceding  status  parameter  (this  feature  permits  the
{       creation of a single status parameter from separate pieces of data).
{
{ TEXT:  (input) This parameter specifies the status parameter text. Trailing
{       spaces in this text are not included in the status record.
{
{ STATUS: (input, output) This parameter specifies the status record to  which
{       the text is to be appended.
{

 PROCEDURE [XREF] osp$append_status_parameter ALIAS 'ospasp'
   (    delimiter: char;
        text: string ( * <= osc$max_string_size);
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc osc$status_parameter_delimiter
*copyc ost$status
*copyc ost$string
?? POP ??
```

14.7.3 clp$append_status_string

```
{
{   This request appends a string as a status parameter to the text field of a
{ status record. This request assumes that the status record has been
{ initialized by a call to osp$set_status_abnormal.
{
{   This request differs from osp$append_status_parameter in that it accepts a
{ string of any length. If the string is too long to fit into the status
{ record's text field, it is truncated on the right.
{
{       CLP$APPEND_STATUS_STRING (DELIMITER, TEXT, STATUS)
{
{ DELIMITER: (input)  This parameter specifies the character that will precede
{       the text supplied by this request in the text field of the status being
{       formed. Depending on its value, this character may cause one of two
{       effects when the status is formatted into a message. If this character
{       is the same as the first character in the text field, then the new text
{       will be treated as a separate status parameter. If, however, this
{       character is different from the first character in the text field, then
{       this character and the new text will be treated as part of the
```

```
{       preceding status parameter (this feature permits the creation of a
{       single status parameter from separate pieces of data).
{
{ TEXT: (input)  This parameter specifies the string text. Trailing spaces in
{       this text are not included in the status record.
{
{ STATUS: (input, output) This parameter specifies the status record to  which
{       the text is to be appended.
{

 PROCEDURE [XREF] clp$append_status_string
   (    delimiter: char;
        text: string ( * );
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc osc$status_parameter_delimiter
*copyc ost$status
?? POP ??
```

14.7.4 osp$append_status_file

```
{
{   The purpose of this request is to append the path name for  a  file  as  a
{ status  parameter  to the text of a status record. Some editing of the path
{ name may be done prior to appending it to the  status  record  in  order  to
{ obtain  the minimal sized representation of the path name, assuming that the
{ status will be interpreted within the current  job.  This  request  assumes
{ that   the   status   record   has   been   initialized   by   a   call   to
{ osp$set_status_abnormal.
{
{       OSP$APPEND_STATUS_FILE (DELIMITER, FILE, STATUS)
{
{ DELIMITER: (input) This parameter specifies the character that will  precede
{       the path name supplied by this request in the text field of the status
{       being formed. Depending on its value, this character may cause one of
{       two  effects  when  the  status  is formatted into a message. If this
{       character is the same as the first character in the text  field,  then
{       the  integer  will  be  treated  as  a separate status parameter. If,
{       however, this character is different from the first character  in  the
{       text  field,  then this character and the path name will be treated as
{       part of the preceding  status  parameter  (this  feature  permits  the
{       creation of a single status parameter from separate pieces of data).
{
{ FILE: (input)  This parameter specifies the path name (file) to be appended.
{
{ STATUS: (input, output) This parameter specifies the status record to  which
{       the text is to be appended.
{

 PROCEDURE [XREF] osp$append_status_file
   (    delimiter: char;
        file: fst$file_reference;
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc amt$local_file_name
*copyc fst$file_reference
*copyc osc$status_parameter_delimiter
```

```
*copyc ost$status
?? POP ??
```

14.7.5 osp$append_status_integer

```
{
{   The purpose of this request  is  to  convert  an  integer  to  its  string
{ representation and append that string as a status parameter to the text of a
{ status record. This  request  assumes  that  the  status  record  has  been
{ initialized by a call to osp$set_status_abnormal.
{
{       OSP$APPEND_STATUS_INTEGER (DELIMITER, INT, RADIX,
{         INCLUDE_RADIX_SPECIFIER, STATUS)
{
{ DELIMITER:  (input) This parameter specifies the character that will precede
{       the integer supplied by this request in the text field of  the  status
{       being formed. Depending on its value, this character may cause one of
{       two effects when the status is formatted  into  a  message.  If  this
{       character  is  the same as the first character in the text field, then
{       the integer will be treated  as  a  separate  status  parameter.  If,
{       however,  this  character is different from the first character in the
{       text field, then this character and the integer  will  be  treated  as
{       part  of  the  preceding  status  parameter  (this feature permits the
{       creation of a single status parameter from separate pieces of data).
{
{ INT: (input) This parameter specifies the integer to be converted.
{
{ RADIX: (input) This parameter specifies the radix  in  which  the  integer's
{       value is to be represented.
{
{ INCLUDE_RADIX_SPECIFIER:   (input)  This  parameter  specifies  whether  the
{       representation of the radix is to be included in the resulting  string
{       -- e.g. (16) for a number with a radix of 16.
{
{ STATUS:  (input, output) This parameter specifies the status record to which
{       the text is to be appended.
{

 PROCEDURE [XREF] osp$append_status_integer ALIAS 'ospasi'
   (    delimiter: char;
        int: integer;
        radix: 2 .. 16;
        include_radix_specifier: boolean;
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc osc$status_parameter_delimiter
*copyc ost$status
?? POP ??
```

14.7.6 osp$append_status_real

```
{
{   This request converts a real  number  to  its  string  representation  and
{ appends  that  string  as  a  status parameter to the text field of a status
{ record. This request assumes that the status record has been initialized by
{ a call to osp$set_status_abnormal.
{
```

```
{       OSP$APPEND_STATUS_REAL (DELIMITER, REAL_NUMBER, NUMBER_OF_DIGITS,
{         STATUS)
{
{ DELIMITER:  (input) This parameter specifies the character that will precede
{       the real number supplied by this request in  the  text  field  of  the
{       status being formed. Depending on its value, this character may cause
{       one of two effects when the status is formatted into  a  message.  If
{       this  character  is the same as the first character in the text field,
{       then the integer will be treated as a separate status parameter.  If,
{       however,  this  character is different from the first character in the
{       text field, then this character and the real number  will  be  treated
{       as  part  of  the preceding status parameter (this feature permits the
{       creation of a single status parameter from separate pieces of data).
{
{ REAL_NUMBER: (input) This is the real number to be appended.
{
{ NUMBER_OF_DIGITS: (input) This specifies the maximum  number  of  digits  to
{       include in the result. There may be fewer digits than this since zero
{       digits to the right of the decimal point are not included.
{
{ STATUS:  (input, output) This parameter specifies the status record to which
{       the text is to be appended.
{

 PROCEDURE [XREF] osp$append_status_real
   (    delimiter: char;
        real_number: longreal;
        number_of_digits: clt$real_number_digit_count;
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$real_number_digit_count
*copyc ost$status
?? POP ??
```

14.7.7 clp$append_status_type

```
{
{   This request appends the string representation of a clt$type_specification
{ as  a  status  parameter to the text field of a status record. This request
{ assumes  that  the  status  record  has  been  initialized  by  a  call  to
{ osp$set_status_abnormal.
{
{       CLP$APPEND_STATUS_TYPE (DELIMITER, TYPE_SPECIFICATION, STATUS)
{
{ DELIMITER:  (input) This parameter specifies the character that will precede
{       the type specification supplied by this request in the text  field  of
{       the  status  record  being  formed.  Depending  on  its  value,  this
{       character may cause one of two effects when the  status  is  formatted
{       into  a message. If this character is the same as the first character
{       in the text field, then the type specification will be  treated  as  a
{       separate  status  parameter. If, however, this character is different
{       from the first character in the text field, then  this  character  and
{       the type specification will be treated as part of the preceding status
{       parameter (this feature  permits  the  creation  of  a  single  status
{       parameter from separate pieces of data).
{
{ TYPE_SPECIFICATION:  (input) This parameter specifies the type specification
{       to be appended.
```

```
{
{ STATUS: (input, output) This parameter specifies the status record to  which
{       the type specification is to be appended.
{

 PROCEDURE [XREF] clp$append_status_type
   (    delimiter: char;
        type_specification: ^clt$type_specification;
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$type_specification
*copyc ost$status
?? POP ??
```

14.7.8 clp$append_status_value_type

```
{
{   This  request  appends  the  string  representation  of  the  type  of   a
{ clt$data_value  as  a status parameter to the text field of a status record.
{ This request assumes that the status record has been initialized by  a  call
{ to osp$set_status_abnormal.
{
{       CLP$APPEND_STATUS_VALUE_TYPE (DELIMITER, VALUE, STATUS)
{
{ DELIMITER:  (input) This parameter specifies the character that will precede
{       the value type supplied by this request  in  the  text  field  of  the
{       status  record  being  formed. Depending on its value, this character
{       may cause one of two effects when  the  status  is  formatted  into  a
{       message.  If this character is the same as the first character in the
{       text field, then the value type will be treated as a  separate  status
{       parameter.  If,  however,  this character is different from the first
{       character in the text field, then this character and  the  value  type
{       will  be  treated  as  part  of  the  preceding status parameter (this
{       feature permits  the  creation  of  a  single  status  parameter  from
{       separate pieces of data).
{
{ VALUE:  (input)  This  parameter  specifies  the  value  whose type is to be
{       appended.
{
{ STATUS: (input, output) This parameter specifies the status record to  which
{       the type specification is to be appended.
{

 PROCEDURE [XREF] clp$append_status_value_type
   (    delimiter: char;
        value: ^clt$data_value;
    VAR status {input, output} : ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$data_value
*copyc ost$status
?? POP ??
```

14.7.9 osp$set_status_from_condition

```
{
{   The purpose of this request is to construct an abnormal status record from
```

```
{ the information presented to a pmt$condition_handler procedure. Attempts to
{ construct a status record for  condition  selectors  pmc$all_conditions  and
{ pmc$condition_combination are in error.
{
{       OSP$SET_STATUS_FROM_CONDITION (IDENTIFIER, CONDITION, SAVE_AREA,
{         CONDITION_STATUS, STATUS)
{
{ IDENTIFIER:  (input)  This  parameter  is  accepted  for  compatibility with
{       previous system versions but is ignored.
{
{ CONDITION: (input) This parameter specifies the  condition  from  which  the
{       status record is to be constructed.
{
{ SAVE_AREA:  (input)  This  parameter  specifies  the  stack  frame save area
{       associated with the condition.
{
{ CONDITION_STATUS: (output) This parameter specifies the  constructed  status
{       record.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITION: ose$invalid_condition_selector, ose$empty_system_condition,
{            ose$empty_block_exit_reason, ose$unknown_segment_condition,
{            ose$invalid_save_area, ose$unknown_interactive_cond
{

 PROCEDURE [XREF] osp$set_status_from_condition (identifier:
   ost$status_identifier,
   condition: pmt$condition;
   save_area: ^ost$stack_frame_save_area;
   VAR condition_status: ost$status;
   VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc OST$STATUS_IDENTIFIER
*copyc PMT$CONDITION
*copyc OST$STACK_FRAME_SAVE_AREA
*copyc OST$STATUS
*copyc OSE$CONDITION_EXCEPTIONS
?? POP ??
```

14.7.10 osp$get_status_severity

```
{
{   The purpose of this request is to return the status severity level for  a
{ status condition.
{
{   The  libraries  in  the  command  list are searched until a message module
{ containing a description  for  the  condition  is  found  and  the  severity
{ contained  therein  is  returned.  If  the  condition  is  not found, error
{ severity is returned.
{
{       OSP$GET_STATUS_SEVERITY (CONDITION, SEVERITY, STATUS)
{
{ CONDITION: (input) This parameter specifies the condition code for which the
{       severity is to be returned.
{
{ SEVERITY: (output) This parameter specifies the status severity level.
{
{ STATUS: (output) This parameter specifies the request status.
```

```
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_status_severity ALIAS 'ospgsv'
   (    condition: ost$status_condition;
    VAR severity: ost$status_severity;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_severity
?? POP ??
```

14.7.11 osp$get_diagnostic_severity

```
{
{   The purpose of this request is to return the diagnostic severity level for
{ a  status condition.
{
{   The  libraries  in  the  command  list are searched until a message module
{ containing a description  for  the  condition  is  found  and  the  severity
{ contained  therein  is  returned.  If  the  condition  is  not found, error
{ severity is returned.
{
{       OSP$GET_DIAGNOSTIC_SEVERITY (CONDITION, SEVERITY, STATUS)
{
{ CONDITION: (input) This parameter specifies the condition code for which the
{       severity is to be returned.
{
{ SEVERITY: (output) This parameter specifies the diagnostic severity level.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_diagnostic_severity ALIAS 'ospgsv'
   (    condition: ost$status_condition;
    VAR severity: ost$diagnostic_severity;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$diagnostic_severity
*copyc ost$status
?? POP ??
```

14.7.12 osp$get_status_condition_string

```
{
{   The purpose of this request is to get  the  string  representation  for  a
{ status  condition,  given its code. The returned string will be in the form
{ "XX n" where XX is the "product identifier" portion of  the  condition  code
{ and  n is the numeric portion.
{
{       OSP$GET_STATUS_CONDITION_STRING (CONDITION, STR, STATUS)
{
{ CONDITION:  (input)  This  parameter  specifies  the  code  for  the  status
{       condition.
{
```

```
{ STR:  (output)  This  parameter  specifies  the string representation of the
{       status condition.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_status_condition_string
   (    condition: ost$status_condition_code;
    VAR str: ost$string;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$string
?? POP ??
```

14.7.13 osp$status_condition_code

```
{
{   This function returns a status condition code formed by combining the
{ specified status identifier and status number.
{
{       OSP$STATUS_CONDITION_CODE (IDENTIFIER, NUMBER): CONDITION
{
{ IDENTIFIER: (input)  This parameter specifies the status identifier.
{
{ NUMBER: (input)  This parameter specifies the status number.
{
{ CONDITION: (result)  The function's result specifies the status condition.
{

 FUNCTION [INLINE] osp$status_condition_code
   (    identifier: ost$status_identifier;
        number: ost$status_condition_number): ost$status_condition_code;

?? PUSH (LISTEXT := ON) ??
*copyc ost$status_condition_code
*copyc ost$status_condition_number
*copyc ost$status_identifier
?? POP ??
```

14.7.14 osp$status_condition_number

```
{
{   This function returns the number component of a status condition code.
{
{       OSP$STATUS_CONDITION_NUMBER (CONDITION): NUMBER
{
{ CONDITION: (input)  This parameter specifies the status condition.
{
{ NUMBER: (result)  The function's result specifies the status number.
{

 FUNCTION [INLINE] osp$status_condition_number
   (    condition: ost$status_condition_code): ost$status_condition_number;
```

```
?? PUSH (LISTEXT := ON) ??
*copyc ost$status_condition_code
*copyc ost$status_condition_number
?? POP ??
```

14.7.15 osp$unpack_status_conditionr

```
{
{   This procedure returns the identifier and number components of
{ a status condition code.
{
{       OSP$UNPACK_STATUS_CONDITION (CONDITION, IDENTIFIER, NUMBER);
{
{ CONDITION: (input)  This parameter specifies the status condition.
{
{ IDENTIFIER: (output)  This parameter specifies the status identifier.
{
{ NUMBER: (output)  This parameter specifies the status number.
{

 PROCEDURE [INLINE] osp$unpack_status_condition
   (    condition: ost$status_condition_code;
    VAR identifier: ost$status_identifier;
    VAR number: ost$status_condition_number);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status_condition_code
*copyc ost$status_condition_number
*copyc ost$status_identifier
?? POP ??
```

14.7.16 osp$unpack_status_identifier

```
{
{   This procedure returns the identifier component of a status condition
{ code.
{
{       OSP$UNPACK_STATUS_IDENTIFIER (CONDITION, IDENTIFIER)
{
{ CONDITION: (input)  This parameter specifies the status condition.
{
{ IDENTIFIER: (output)  This parameter specifies the status identifier.
{

 PROCEDURE [INLINE] osp$unpack_status_identifier
   (    condition: ost$status_condition_code;
    VAR identifier: ost$status_identifier);

?? PUSH (LISTEXT := ON) ??

 PROEND osp$unpack_status_identifier;

*copyc ost$status_condition_code
*copyc ost$status_identifier
?? POP ??
```

14.7.17 osp$format_message

```
{
{   The purpose of this request is to return a  message  that  represents  the
{ status  condition.  The  message is returned in a sequence which contains a
{ ost$status_message_line_count followed by that  many  message  lines.  Each
{ message line consists of a ost$status_message_line_size followed by a string
{ of that size. These lines  can  be  written  to  a  file  or  log  via  the
{ appropriate interfaces. Each line begins with a space character to serve as
{ a "format effector" in the event the formatted message is written to a  list
{ file.
{
{   The  template  for  the status message is located by searching each object
{ library in the command list until a message module  containing  the  desired
{ template  is  found.  If  such  a  module is found but the module's natural
{ language is  not  the  one  currently  selected  for  the  job,  the  search
{ continues.  If  no template in the desired natural language is found, but a
{ template in the default natural language (US_English) was found, it is used.
{ If still no template has been picked, a "default template" is used which has
{ the effect of displaying the "raw" status record.
{
{       OSP$FORMAT_MESSAGE (MESSAGE_STATUS, MESSAGE_LEVEL, MAX_MESSAGE_LINE,
{         MESSAGE, STATUS)
{
{ MESSAGE_STATUS: (input) This parameter specifies the status to be formatted.
{       The condition field is used to locate a template for the message. The
{       message is  then  formatted  under  control  of  this  template.  The
{       template  may  "request" information from the text field of the status
{       to be substituted into the formatted message. To accomodate this, the
{       first  character  in  the  text  field  defines  the  status parameter
{       delimiter for the status record, i.e. each sequence of characters  in
{       the text field that is preceded by the status parameter delimiter is a
{       separate status parameter that may be substituted into  the  formatted
{       message.
{
{ MESSAGE_LEVEL: (input) This parameter specifies the level for the message to
{       be formatted.
{
{ MAX_MESSAGE_LINE: (input) This parameter specifies  the  maximum  number  of
{       characters that can be placed in a line produced by this request. The
{       message formatter will try to "break" long lines at a  delimiter;  but
{       if  this  cannot  be  done,  two dots will be placed at the end of the
{       "broken" line to mark its continuation.
{
{ MESSAGE: (output) This parameter specifies the container for  the  formatted
{       message.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$format_message ALIAS 'ospfmsg'
   (    message_status: ost$status;
        message_level: ost$format_message_level;
        max_message_line: ost$max_status_message_line;
    VAR message: ost$status_message;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
```

```
*copyc ost$format_message_level
*copyc ost$max_status_message_line
*copyc ost$status
*copyc ost$status_message
?? POP ??
```

14.7.18 osp$get_status_condition_name

```
{
{   The purpose of this request is to get the name  for  a  status  condition,
{ given its code.
{
{   The  libraries  in  the  command  list are searched until a message module
{ containing a description for the  condition  code  is  found  and  the  name
{ contained  therein  is  returned.  If  the  code  is  not  found,  the name
{ UNKNOWN_CONDITION is returned.
{
{       OSP$GET_STATUS_CONDITION_NAME (CODE, NAME, STATUS)
{
{ CODE: (input) This parameter specifies the code for the status condition.
{
{ NAME: (output) This parameter specifies the name of the status condition.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_status_condition_name
   (    code: ost$status_condition_code;
    VAR name: ost$status_condition_name;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$status_condition_name
?? POP ??
```

14.7.19 osp$get_status_condition_code

```
{
{   The purpose of this request is to get the code  for  a  status  condition,
{ given its name.
{
{   The  libraries  in  the  command  list are searched until a message module
{ containing a description for the  condition  name  is  found  and  the  code
{ contained  therein is returned. If the name is not found, a code of zero is
{ returned.
{
{       OSP$GET_STATUS_CONDITION_CODE (NAME, CODE, STATUS)
{
{ NAME: (input) This parameter specifies the name of the status condition.
{
{ CODE: (output) This parameter specifies the code for the status condition.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{
```

```
 PROCEDURE [XREF] osp$get_status_condition_code
   (    name: ost$status_condition_name;
    VAR code: ost$status_condition_code;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$status_condition_name
?? POP ??
```

14.7.20 osp$get_message_level

```
{
{   The purpose of this request is to get the message level currently  in  use
{ in the job.
{
{       OSP$GET_MESSAGE_LEVEL (MESSAGE_LEVEL, STATUS)
{
{ MESSAGE_LEVEL: (output)  This parameter specifies the current message level.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_message_level
   (VAR message_level: ost$status_message_level;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$status
*copyc ost$status_message_level
?? POP ??
```

14.7.21 osp$set_message_level

```
{
{   The purpose of this request is to set the message level for the job.
{
{       OSP$SET_MESSAGE_LEVEL (MESSAGE_LEVEL, STATUS)
{
{ MESSAGE_LEVEL: (output) This parameter specifies the new message level.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: ose$bad_message_level
{

 PROCEDURE [XREF] osp$set_message_level
   (    message_level: ost$status_message_level;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ose$message_gen_exceptions
*copyc ost$status
*copyc ost$status_message_level
?? POP ??
```

14.7.22 osp$get_natural_language

```
{
{   The purpose of this request is to get the natural  language  currently  in
{ use in the job.
{
{       OSP$GET_NATURAL_LANGUAGE (NATURAL_LANGUAGE, STATUS)
{
{ NATURAL_LANGUAGE:  (output)  This  parameter  specifies  the current natural
{       language.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_natural_language
   (VAR natural_language: ost$natural_language;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$natural_language
*copyc ost$status
?? POP ??
```

14.7.23 osp$set_natural_language

```
{
{   The purpose of this request is to set the natural language for the job.
{
{       OSP$SET_NATURAL_LANGUAGE (NATURAL_LANGUAGE, STATUS)
{
{ NATURAL_LANGUAGE:  (output)  This  parameter  specifies  the   new   natural
{       language.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: ose$bad_natural_language
{

 PROCEDURE [XREF] osp$set_natural_language
   (    natural_language: ost$natural_language;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ose$message_gen_exceptions
*copyc ost$natural_language
*copyc ost$status
?? POP ??
```

14.7.24 osp$get_status_message_by_code

```
{
{   The purpose of this request is to get the name, status severity,
{ diagnostic severity and message template for a status condition code.
{
{   The libraries in the command list are searched for a message module
{ containing the desired condition code. If such a module is found but the
{ module's natural language is not the one currently selected for the job, the
{ search continues. If no module in the desired natural language is found,
```

```
{ but a module in the default natural language (US_English) was found, it is
{ used. If still no information for the condition code can be found, defaults
{ are returned (see the individual parameter descriptions, below).
{
{       OSP$GET_STATUS_MESSAGE_BY_CODE (CONDITION_CODE, CONDITION_NAME,
{         STATUS_SEVERITY, DIAGNOSTIC_SEVERITY, MESSAGE_TEMPLATE, STATUS)
{
{ CONDITION_CODE: (input)  This parameter specifies the condition code to be
{       searched for.
{
{ CONDITION_NAME: (output)  This parameter specifies the name of the
{       condition. If the condition code could not be found, the name
{       'UNKNOWN_CONDITION' is returned.
{
{ STATUS_SEVERITY: (output)  This parameter specifies the status severity of
{       the condition. If the condition code could not be found,
{       OSC$ERROR_STATUS is returned.
{
{ DIAGNOSTIC_SEVERITY: (output)  This parameter specifies the diagnostic
{       severity of the condition. If the condition code could not be found,
{       OSC$ERROR_SEVERITY is returned.
{
{ MESSAGE_TEMPLATE: (output)  This parameter specifies the pointer to the
{       message template for the condition. If the condition code could not
{       be found, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS: none
{

 PROCEDURE [XREF] osp$get_status_message_by_code
   (    condition_code: ost$status_condition_code;
    VAR condition_name: ost$status_condition_name;
    VAR status_severity: ost$status_severity;
    VAR diagnostic_severity: ost$diagnostic_severity;
    VAR message_template: ^ost$message_template;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$diagnostic_severity
*copyc ost$message_template
*copyc ost$status
*copyc ost$status_condition_code
*copyc ost$status_condition_name
*copyc ost$status_severity
?? POP ??
```

14.8 Status and Message Type Declarations

14.8.1 osc$max_condition

```
 CONST
   osc$max_condition = osc$max_status_condition_code;

*copyc osc$max_status_condition_code
```

14.8.2 osc$max_status_condition_code

```
 CONST
   osc$max_status_condition_code = 0fffffffff(16);
```

14.8.3 osc$max_status_condition_number

```
 CONST
   osc$max_status_condition_number = 0ffffff(16);
```

14.8.4 osc$max_status_message

```
 CONST
   osc$max_status_message = osc$status_message_height *
         osc$status_message_width;

*copyc osc$status_message_height
*copyc osc$status_message_width
```

14.8.5 osc$max_status_message_line

```
 CONST
   osc$max_status_message_line = 1 + osc$max_status_message;

*copyc osc$max_status_message
```

14.8.6 osc$max_status_message_lines

```
 CONST
   osc$max_status_message_lines = osc$max_status_message;

*copyc osc$max_status_message
```

14.8.7 osc$min_status_message_line

```
 CONST
   osc$min_status_message_line = 1 + osc$max_name_size;

*copyc ost$name
```

14.8.8 osc$status_message_height

```
 CONST
   osc$status_message_height = 30;
```

14.8.9 osc$status_message_width

```
 CONST
   osc$status_message_width = 132;
```

14.8.10 osc$status_parameter_delimiter

```
 CONST
   osc$status_parameter_delimiter = $CHAR (31) {Unit Separator} ;
```

14.8.11 ost$diagnostic_severity

```
 TYPE
   ost$diagnostic_severity = (osc$non_standard_severity,
         osc$dependent_severity, osc$informative_severity,
         osc$warning_severity, osc$error_severity, osc$fatal_severity,
         osc$catastrophic_severity);
```

14.8.12 ost$format_message_level

```
 TYPE
   ost$format_message_level = (osc$current_message_level,
         osc$brief_message_level, osc$full_message_level);
```

14.8.13 ost$max_status_message_line

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

*copyc osc$max_status_message_line
*copyc osc$min_status_message_line
```

14.8.14 ost$natural_language

```
 TYPE
   ost$natural_language = ost$name;

?? FMT (FORMAT := OFF) ??

 CONST
   osc$danish                     = 'DANISH                      ',
   osc$dutch                      = 'DUTCH                       ',
   osc$english                    = 'ENGLISH                     ',
   osc$finnish                    = 'FINNISH                     ',
   osc$flemish                    = 'FLEMISH                     ',
   osc$french                     = 'FRENCH                      ',
   osc$german                     = 'GERMAN                      ',
   osc$italian                    = 'ITALIAN                     ',
   osc$norwegian                  = 'NORWEGIAN                   ',
   osc$portuguese                 = 'PORTUGUESE                  ',
   osc$spanish                    = 'SPANISH                     ',
   osc$swedish                    = 'SWEDISH                     ',
   osc$us_english                 = 'US_ENGLISH                  ',

   osc$default_natural_language   = osc$us_english,

   osc$current_natural_language   = osc$null_name;

?? FMT (FORMAT := ON) ??
```

```
*copyc ost$name
```

14.8.15 ost$status

```
 TYPE
   ost$status = record
     case normal: boolean of
     = FALSE =
       condition: ost$status_condition_code,
       text: ost$string,
     = TRUE =
       ,
     casend,
   recend;

*copyc osc$max_condition
*copyc osc$status_parameter_delimiter
*copyc ost$status_condition
*copyc ost$status_condition_code
*copyc ost$string
```

14.8.16 ost$status_condition

```
 TYPE
   ost$status_condition = ost$status_condition_code;

*copyc ost$status_condition_code
```

14.8.17 ost$status_condition_code

```
 TYPE
   ost$status_condition_code = 0 .. osc$max_status_condition_code;

*copyc osc$max_status_condition_code
```

14.8.18 ost$status_condition_name

```
 TYPE
   ost$status_condition_name = ost$name;

*copyc ost$name
```

14.8.19 ost$status_condition_number

```
 TYPE
   ost$status_condition_number = 0 .. osc$max_status_condition_number;

*copyc osc$max_status_condition_number
```

14.8.20 ost$status_identifier

```
 TYPE
   ost$status_identifier = string (2);
```

14.8.21 ost$status_message

```
TYPE
  ost$status_message = SEQ (ost$status_message_line_count,
        {} REP osc$status_message_height of ost$status_message_line_size,
        {} REP osc$status_message_height * (1 + osc$status_message_width) of
        char);

*copyc osc$status_message_height
*copyc osc$status_message_width
*copyc ost$status_message_line
*copyc ost$status_message_line_count
*copyc ost$status_message_line_size
```

14.8.22 ost$status_message_level

```
TYPE
  ost$status_message_level = osc$brief_message_level ..
    osc$full_message_level;

*copyc ost$format_message_level
```

14.8.23 ost$status_message_line

```
TYPE
  ost$status_message_line = string ( * <= osc$max_status_message_line);

*copyc osc$max_status_message_line
```

14.8.24 ost$status_message_line_count

```
TYPE
  ost$status_message_line_count = 0 .. osc$max_status_message_lines;

*copyc osc$max_status_message_lines
```

14.8.25 ost$status_message_line_size

```
TYPE
  ost$status_message_line_size = 0 .. osc$max_status_message_line;

*copyc osc$max_status_message_line
```

14.8.26 ost$status_severity

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

14.8.27 ost$diagnostic_severity

```
TYPE
  ost$diagnostic_severity = (osc$non_standard_severity,
        osc$dependent_severity, osc$informative_severity,
```

```
osc$warning_severity, osc$error_severity, osc$fatal_severity,
osc$catastrophic_severity);
```

15 Help Message Services

This section describes the interfaces used by the SCL interpreter and other applications to retrieve prompts, menus and help messages stored in “help modules” on object libraries.

The information retrieved and used by these requests is created via the CREATE_MESSAGE_MODULE subutility within the CREATE_OBJECT_LIBRARY utility.

An application begins use of these interfaces with OSP$FIND_HELP_MODULE which returns a pointer to the application's help module. The remaining requests provide for locating specific information within the help module. This information is of two general types: application menus, and messages or prompts.

The information returned by OSP$FIND_APPLICATION_MENU is passed on, by the application, to the CSP$SET_MENU request.

A message or prompt returned by the other OSP$FIND_xxx intrerfaces is passed on to OSP$FORMAT_HELP_MESSAGE, the result of which is written to some device or file using appropriate requests.

15.1 Help Message Services Interfaces

15.1.1 osp$find_help_module

```
{
{   The purpose of this request is to search for a help module. Help  modules
{ reside  on object libraries and contain prompts and help information for SCL
{ commands and functions, and descriptions of menus for applications. A  help
{ module may also contain status message descriptions.
{
{   A help module is located by searching each object library in  the  command
{ list for the name of the help module. The name of the help module is formed
{ by using the seed name passed to this request and suffixing it with a dollar
{ sign  ($)  character  followed by the name of the preferred natural language
{ selected for the job. If no module with the resulting name  is  found,  the
{ process  is  repeated,  this  time  using  the  name  of the default natural
{ language (US_English).
{
{       OSP$FIND_HELP_MODULE (SEED_NAME, HELP_MODULE, ONLINE_MANUAL_NAME,
{         NATURAL_LANGUAGE, STATUS)
{
{ SEED_NAME: (input) This parameter specifies the name that is to be  suffixed
{       with  a $ and the name of the natural language to form the name of the
{       module to be searched for.
{
{ HELP_MODULE: (output) This parameter  specifies  the  pointer  to  the  help
{       module.  This  pointer  is  used  as  input  to  requests that locate
{       specific messages within  the  help  module.  If  this  parameter  is
{       returned as NIL, the specified help module could not be found.
{
{ ONLINE_MANUAL_NAME: (output) This parameter specifies the name of the online
{       manual associated with the help module. If the specified help  module
{       could not be found or if there is no online manual associated with the
{       help module, osc$null_name is returned.
{
{ NATURAL_LANGUAGE: (output) This parameter specifies the name of the  natural
{       language  in  which  the  messages, prompts and menus contained in the
{       help module are composed. If the specified help module could  not  be
{       found, osc$null_name is returned.
```

```
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_help_module
   (    seed_name: pmt$program_name;
    VAR help_module: ^ost$help_module;
    VAR online_manual_name: ost$online_manual_name;
    VAR natural_language: ost$natural_language;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$help_module
*copyc ost$name
*copyc ost$natural_language
*copyc ost$online_manual_name
*copyc ost$status
*copyc pmt$program_name
?? POP ??
```

15.1.2 osp$find_application_menu

```
{
{   The purpose of this request is to find a particular menu  description  for
{ an application within a help module.
{
{       OSP$FIND_APPLICATION_MENU (HELP_MODULE, MENU_NAME, MENU_CLASSES,
{         MENU_ITEMS, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ MENU_NAME: (input) This parameter specifies the name of the application menu
{       to be found.
{
{ MENU_CLASSES:   (output)   This  parameter  specifies  the  pointer  to  the
{       definition of the menu classes. If the specified menu  could  not  be
{       found in the help module, NIL is returned.
{
{ MENU_ITEMS:  (output) This parameter specifies the pointer to the definition
{       of the menu items. If the specified menu could not be  found  in  the
{       help module, NIL is returned.
{
{  STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_application_menu
   (    help_module: ^ost$help_module;
        menu_name: ost$application_menu_name;
    VAR menu_classes: cst$menu_class;
    VAR menu_items: cst$menu_list;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc cst$menu_class
*copyc cst$menu_list
*copyc ost$application_menu_name
```

```
*copyc ost$help_module
*copyc ost$status
?? POP ??
```

15.1.3 osp$find_brief_help_message

```
{
{   The purpose of this request is to find the template  for  the  brief  help
{ message for an SCL command or function within a help module.
{
{       OSP$FIND_BRIEF_HELP_MESSAGE (HELP_MODULE, MESSAGE_TEMPLATE, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ MESSAGE_TEMPLATE: (output) This  parameter  specifies  the  pointer  to  the
{       template  for the message. If no brief help message could be found in
{       the help module, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_brief_help_message
   (    help_module: ^ost$help_module;
    VAR message_template: ^ost$message_template;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$help_module
*copyc ost$message_template
*copyc ost$status
?? POP ??
```

15.1.4 osp$find_full_help_message

```
{
{   The purpose of this request is to find the  template  for  the  full  help
{ message for an SCL command or function within a help module.
{
{       OSP$FIND_FULL_HELP_MESSAGE (HELP_MODULE, MESSAGE_TEMPLATE, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ MESSAGE_TEMPLATE: (output) This  parameter  specifies  the  pointer  to  the
{       template  for  the message. If no full help message could be found in
{       the help module, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_full_help_message
   (    help_module: ^ost$help_module;
    VAR message_template: ^ost$message_template;
    VAR status: ost$status);
```

```
?? PUSH (LISTEXT := ON) ??
*copyc ost$help_module
*copyc ost$message_template
*copyc ost$status
?? POP ??
```

15.1.5 osp$find_parameter_prompt

```
{
{   The purpose of this request is to find the template for the prompt  for  a
{ parameter for an SCL command or function within a help module.
{
{       OSP$FIND_PARAMETER_PROMPT (HELP_MODULE, PARAMETER_NAME,
{         PROMPT_TEMPLATE, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ PARAMETER_NAME: (input) This parameter specifies the name of the command  or
{       function parameter whose prompt is to be found.
{
{ PROMPT_TEMPLATE:  (output)  This  parameter  specifies  the  pointer  to the
{       template for the prompt. If no prompt  for  the  parameter  could  be
{       found in the help module, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_parameter_prompt
   (    help_module: ^ost$help_module;
        parameter_name: clt$parameter_name;
    VAR prompt_template: ^ost$message_template;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_name
*copyc ost$help_module
*copyc ost$message_template
*copyc ost$status
?? POP ??
```

15.1.6 osp$find_param_assist_prompt

```
{
{   The purpose of this request is to find the  template  for  the  assistance
{ prompt for  a parameter for an SCL command or function within a help module.
{
{       OSP$FIND_PARAM_ASSIST_PROMPT (HELP_MODULE, PARAMETER_NAME,
{         PROMPT_TEMPLATE, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ PARAMETER_NAME:  (input) This parameter specifies the name of the command or
{       function parameter whose assistance prompt is to be found.
{
{ PROMPT_TEMPLATE: (output)  This  parameter  specifies  the  pointer  to  the
```

```
{       template  for  the  assistance prompt. If no prompt for the parameter
{       could be found in the help module, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_param_assist_prompt
   (    help_module: ^ost$help_module;
        parameter_name: clt$parameter_name;
    VAR prompt_template: ^ost$message_template;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_name
*copyc ost$help_module
*copyc ost$message_template
*copyc ost$status
?? POP ??
```

15.1.7 osp$find_parameter_help_message

```
{
{   The purpose of this request is to find the template for the  help  message
{ for a parameter for an SCL command or function within a help module.
{
{       OSP$FIND_PARAMETER_HELP_MESSAGE (HELP_MODULE, PARAMETER_NAME,
{         MESSAGE_TEMPLATE, STATUS)
{
{ HELP_MODULE: (input) This parameter specifies the pointer to the help module
{       that is to be searched.
{
{ PARAMETER_NAME: (input) This parameter specifies the name of the command  or
{       function parameter whose help message is to be found.
{
{ MESSAGE_TEMPLATE:  (output)  This  parameter  specifies  the  pointer to the
{       template for the message. If no help message for the parameter  could
{       be found in the help module, NIL is returned.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$find_parameter_help_message
   (    help_module: ^ost$help_module;
        parameter_name: clt$parameter_name;
    VAR message_template: ^ost$message_template;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc clt$parameter_name
*copyc ost$help_module
*copyc ost$message_template
*copyc ost$status
?? POP ??
```

15.1.8 osp$format_help_message

```
{
{   The purpose of this request is to format a message according to a template
{ for  that  message.  The  template  consists of message text and formatting
{ codes that direct the formatting  process.  All  of  the  formatting  codes
{ available  for  status messages are also available to the messages processed
{ by this request. Parameters to be substituted into the message are supplied
{ via  an  array of pointers to strings, each string representing a particular
{ message parameter.
{
{   The    message    is    returned    in    a    sequence    which    contains    a
{ ost$status_message_line_count followed by that  many  message  lines.  Each
{ message line consists of a ost$status_message_line_size followed by a string
{ of that size. These lines can be written to  a  file  via  the  appropriate
{ interfaces.  Each  line begins with a space character to serve as a "format
{ effector" in the event the formatted message is written to a list file.
{
{       OSP$FORMAT_HELP_MESSAGE (MESSAGE_TEMPLATE, MESSAGE_PARAMETERS,
{         MAX_MESSAGE_LINE, MESSAGE, STATUS)
{
{ MESSAGE_TEMPLATE: (input) This parameter  specifies  the  template  for  the
{       message to be formatted.
{
{ MESSAGE_PARAMETERS: (input) This parameter specifies the items of text to be
{       substituted  into  the message according to the parameter substitution
{       formatting codes in the message template. If  NIL  is  specified  for
{       this  parameter,  all  message  parameters are considerred to be null.
{       If NIL is specified for any  particular  element  of  the  array,  the
{       corresponding message parameter is considerred to be null.
{
{ MAX_MESSAGE_LINE:  (input)  This  parameter  specifies the maximum number of
{       characters that can be placed in a line produced by this request. The
{       message  formatter  will try to "break" long lines at a delimiter; but
{       if this cannot be done, two dots will be placed  at  the  end  of  the
{       "broken" line to mark its continuation.
{
{ MESSAGE:  (output)  This parameter specifies the container for the formatted
{       message.
{
{ STATUS: (output) This parameter specifies the request status.
{       CONDITIONS:
{

 PROCEDURE [XREF] osp$format_help_message
   (    message_template: ^ost$message_template;
        message_parameters: ^ost$message_parameters;
        max_message_line: ost$max_status_message_line;
    VAR message: ost$status_message;
    VAR status: ost$status);

?? PUSH (LISTEXT := ON) ??
*copyc ost$max_status_message_line
*copyc ost$message_parameters
*copyc ost$message_template
*copyc ost$status
*copyc ost$status_message
?? POP ??
```

15.2 Help Message Services Type Declarations

15.2.1 ost$application_menu_name

```
TYPE
  ost$application_menu_name = ost$name;

*copyc ost$name
```

15.2.2 ost$help_module

```
TYPE
  ost$help_module = SEQ ( * );
```

15.2.3 ost$message_parameter

```
TYPE
  ost$message_parameter = string ( * );
```

15.2.4 ost$message_parameters

```
TYPE
  ost$message_parameters = array [1 .. * ] of ^ost$message_parameter;

*copyc ost$message_parameter
```

15.2.5 ost$online_manual_name

```
TYPE
  ost$online_manual_name = ost$name;

*copyc ost$name
```

15.2.6 csc$max_classes

```
CONST
  csc$max_classes = 16;
```

15.2.7 csc$max_items_per_class

```
CONST
  csc$max_items_per_class = 20;
```

15.2.8 csc$max_menu_items

```
CONST
  csc$max_menu_items = csc$max_classes*csc$max_items_per_class;

*copyc csc$max_classes
*copyc csc$max_items_per_class
```

15.2.9 cst$application_functions

```
CONST
  csc$max_fkeys = 16;

TYPE
  cst$application_functions = (csc$f1, csc$sf1, csc$f2, csc$sf2, csc$f3,
    csc$sf3, csc$f4, csc$sf4, csc$f5, csc$sf5, csc$f6, csc$sf6, csc$f7,
    csc$sf7, csc$f8, csc$sf8, csc$f9, csc$sf9, csc$f10, csc$sf10, csc$f11,
    csc$sf11, csc$f12, csc$sf12, csc$f13, csc$sf13, csc$f14, csc$sf14,
    csc$f15, csc$sf15, csc$f16, csc$sf16);
```

15.2.10 cst$class_name

```
TYPE
  cst$class_name = ost$name;

*copyc ost$name
```

15.2.11 cst$key_type

```
TYPE
  cst$key_type = (csc$standard_function, csc$application_function,
                  csc$screen_function, csc$unused_entry);
```

15.2.12 cst$max_classes

```
TYPE
  cst$max_classes = 0 .. csc$max_classes;

*copyc csc$max_classes
```

15.2.13 cst$menu_class

```
TYPE
  cst$menu_class = ^ARRAY [1 .. * {csc$max_classes}] of cst$class_name;

*copyc cst$class_name
```

15.2.14 cst$menu_item

```
TYPE
 cst$menu_item = recend
   pair_with_previous: boolean,
   short_label: string(6),
   alternate_short_label: string(6),
   long_label: ost$name,
   alternate_long_label: ost$name,
   menu_parent: cst$max_classes,
   item_assigned: boolean,
   case menu_type: cst$key_type of
   = csc$standard_function =
     standard_function: cst$standard_functions,
   = csc$application_function =
     application_function: cst$application_functions,
```

```
    = csc$screen_function =
      screen_function: cst$screen_events,
    = csc$unused_entry =
      ,
    casend,
  recend;

*copyc ost$name
*copyc cst$max_classes
*copyc cst$key_type
*copyc cst$standard_functions
*copyc cst$application_functions
*copyc cst$screen_events
```

15.2.15 cst$menu_item_number

```
TYPE
  cst$menu_item_number = 0 .. csc$max_menu_items;

*copyc csc$max_menu_items
```

15.2.16 cst$screen_events

```
TYPE
  cst$screen_events = (csc$insert_line, csc$delete_line, csc$home, csc$clear,
        csc$insert_char_menu_item, csc$delete_char_menu_item,
        csc$clear_eol_menu_item);
```

15.2.17 cst$standard_functions

```
TYPE
  cst$standard_functions = (csc$next, csc$sh_next, csc$help, csc$sh_help,
    csc$stop, csc$sh_stop, csc$back, csc$sh_back, csc$up, csc$sh_up, csc$down,
    csc$sh_down, csc$forward, csc$sh_forward, csc$backward, csc$sh_backward,
    csc$edit, csc$sh_edit, csc$data, csc$sh_data, csc$undo, csc$sh_undo);
```

15.2.18 cst$menu_list

```
TYPE
cst$menu_list = ^ARRAY [1 .. * {csc$max_menu_items}] of cst$menu_item;

*copyc cst$menu_item
```