Programming with DOMAIN Advanced System Calls

Order No. 008542
Revision 00
Software Release 9.0

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824
Preface

This manual describes newly released DOMAIN system calls. These calls are in addition to those released in the DOMAIN System Call Reference. This manual is intended to provide adequate documentation and syntax information for many existing system calls. This reference material is intended for programmers who have experience programming with DOMAIN system routines and insert files.

This manual is divided into two parts. Part I contains a general overview of a separate operating subsystem (e.g., the Command Line Handler, File and Tree Utility). Most chapters conclude with a sample program in Pascal demonstrating the calls.

Part II consists of reference information similar to the DOMAIN System Call Reference. Each subsystem has its own section, which contains the data types the subsystem uses, the syntax of its programming calls, and the error messages it generates.

Some calls described in this book belong to released subsystems (PM, PROC2, and RWS) that are described in the DOMAIN System Call Reference manual. To make it easy for you to incorporate this information, we repeated the entire sections, marking the new material with revision bars. This way, you can replace the entire section in your copy of the manual.

For easy organization, the sections in Part II are in alphabetical order by subsystem name. We numbered the pages of Part II by subsystem. For example, the third page in the CL section is page CL-3.

You should use this manual with the programming handbooks listed under Related Documents. These programming handbooks give detailed instructions about using these programming calls.

Audience

This manual is intended for advanced programmers who are writing application programs using DOMAIN system calls. Readers of this manual should be familiar with general DOMAIN system calls as described in Programming with General System Calls. They should also have experience with FORTRAN, Pascal, or C and the operating system as described in the DOMAIN System User’s Guide. This manual is not intended as a tutorial document, but as a reference for programmers who need to use operating system calls.

Related Documents

The Programming With General System Calls handbook (005506), documents how to write programs that use standard DOMAIN system calls including the ACLM, CAL, EC2, ERROR, MTS, NAME, PAD, PUBS, PFM, PGM, PM, PROC1, PROC2, RWS, SIO, STREAM, TIME, TONE, TPAD, and VFMT calls.

The Programming With System Calls for Interprocess Communication handbook (005696), documents how to write programs that use the DOMAIN interprocess facilities including the MBX, MS, IPC, MUTEX, and EC2 calls.

The Programming With DOMAIN 2D Graphics Metafile Resource handbook (005097), documents how to write programs that use the DOMAIN 2D Graphics Metafile Resource.

The Programming With DOMAIN Graphic Primitives handbook (005808), documents how to write graphics programs that use the DOMAIN Graphics Primitive Resource.
Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

UPPERCASE

Uppercase words or characters in formats and command descriptions represent keywords that you must use literally.

lowercase

Lowercase words or characters in formats and command descriptions represent values that you must supply.

[ ]

Square brackets enclose optional items.

{ }

Braces enclose a list from which you must choose an item.

| |

A vertical bar separates items in a list of choices.

< >

Angle brackets enclose the name of a key on the keyboard.

CTRL/Z

The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you type the character.

Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

Vertical ellipsis points mean that we have omitted irrelevant parts of a figure or example.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels, you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the DOMAIN System Command Reference manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same description on-line by typing:

$ HELP CRUCR <RETURN>

For documentation comments, a Reader's Response form is located at the back of each manual.
Contents

Part I. Using the DOMAIN Advanced System Calls

Chapter 1 Parsing the Command Line
1.1. System Calls, Insert Files, and Data Types 1-1
1.2. Overview of the CL Manager 1-1
1.3. Initializing the CL Manager
   1.3.1. Defining CL Options 1-4
   1.3.2. Defining Wildcard Options 1-8
   1.3.3. Example of Initializing the CL Manager 1-8
1.4. Getting Flags
   1.4.1. Example of Handling Synonymous Flags with CL_$GET_ENUM_FLAG 1-12
1.5. Reading Arguments
   1.5.1. Getting Arguments Associated with Flags 1-14
   1.5.2. Getting Derived Names 1-15
1.6. Using Miscellaneous CL Calls 1-17
1.7. Sample Program Using CL System Calls 1-18

Chapter 2 Using the File and Tree Utility
2.1. System Calls, Insert Files, and Data Types 2-1
2.2. Overview of the FU System Calls
   2.2.1. Setting FU Options 2-2
   2.2.2. Releasing Storage with FU_$RELEASE_STORAGE 2-6
2.3. Sample Program Using the FU System Calls 2-6

Chapter 3 Logging In and Changing the Registry
3.1. System Calls, Insert Files, and Data Types 3-1
3.2. Overview of the LOGIN System Calls 3-1
3.3. Tailoring a Log-In Operation
   3.3.1. Sample Program Using LOGIN_$LOGIN 3-4
3.4. Changing User Account Files in the Network Registry
   3.4.1. Sample Program — Changing the Registry ACCOUNT Files 3-12

Chapter 4 More Process Manager System Calls
4.1. System Calls, Insert Files, and Data Types 4-1
4.2. Loading and Calling a Program with LOADER System Calls
   4.2.1. Sample Programs Using LOADER System Calls 4-4
4.3. Setting a Process Priority 4-6
4.4. Assigning a Name to a Process 4-6
4.5. Sample Programs Using PM, PROC2 System Calls 4-6

Apollo Confidential
5-1
5.1. System Calls, Insert Files, and Data Types
5-1
5.2. Overview of the RWS System Calls
5-1
5.3. Overview of the BAF System Calls
5-4
5.3.1. Improving Performance Under Current Implementation
5-5
5.4. Sample Program Using RWS and BAF System Calls
5-5

Part II. DOMAIN Advanced System Call Reference

Index

Index-1
Illustrations

Figure 1-1. Structure of a Command Line Token List 1-2
Figure 1-2. Model for Parsing the Command Line 1-3
Figure 3-1. Using LOGIN System Calls in Proper Sequence 3-12
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>System Calls to Initialize the CL Manager</td>
<td>1-4</td>
</tr>
<tr>
<td>1-2</td>
<td>CL Options</td>
<td>1-6</td>
</tr>
<tr>
<td>1-3</td>
<td>Setting Wildcard Options</td>
<td>1-8</td>
</tr>
<tr>
<td>1-4</td>
<td>Miscellaneous CL System Calls</td>
<td>1-17</td>
</tr>
<tr>
<td>2-1</td>
<td>FU System Calls to Operate on Files and Trees</td>
<td>2-2</td>
</tr>
<tr>
<td>2-2</td>
<td>FU Options To Provide Shell Command Options</td>
<td>2-3</td>
</tr>
<tr>
<td>4-1</td>
<td>Difference Between LOADER and PGM_$INVOKE System Calls</td>
<td>4-2</td>
</tr>
<tr>
<td>4-2</td>
<td>PM_$LOAD Options</td>
<td>4-2</td>
</tr>
<tr>
<td>5-1</td>
<td>RWS System Calls to Allocate Dynamic Storage</td>
<td>5-2</td>
</tr>
<tr>
<td>5-2</td>
<td>Summary of Types of Storage Allocation</td>
<td>5-3</td>
</tr>
</tbody>
</table>
Examples

Example 1-1. Redefining the CL Options Set 1-7
Example 1-2. Initializing CL with CL_$SETUP and CL_$PARSE_LINE 1-9
Example 1-3. Initializing CL with CL_$SETUP and CL_$PARSE_INPUT 1-10
Example 1-4. Using CL_$GET_ENUM_FLAG 1-13
Example 1-5. Getting Arguments Associated with Flags 1-15
Example 1-6. Reading Arguments from the Token List 1-16
Example 1-7. Sample Program Using CL System Calls 1-18
Example 2-1. Setting FU Options 2-5
Example 2-2. Operating on Files and Trees with FU Calls 2-6
Example 3-1. Declaring External I/O Routines for LOGIN_$LOGIN 3-2
Example 3-2. Writing LOGIN External I/O Routines for LOGIN_$LOGIN 3-3
Example 3-3. Performing a Log-In Operation with LOGIN_$LOGIN 3-5
Example 3-4. Providing I/O Routines for LOGIN_$LOGIN 3-8
Example 3-5. Using LOGIN Calls to Change ACCOUNT Files 3-14
Example 4-1. Returning a 16-Bit Value with PM_$CALL 4-3
Example 4-2. Loading and Calling a Program 4-4
Example 4-3. Setting Name and Priority of a Process 4-7
Example 5-1. Allocating Storage with BAF System Calls 5-6
Part I. Using the DOMAIN Advanced System Calls
Chapter 1
Parsing the Command Line

The Common Command Line Handler (CL) is a set of DOMAIN system routines that provides an easy and consistent way to read tokens from the command line. CL can perform the following:

- Expand wildcards to existing pathnames
- Handle derived names
- Handle names-file input when the user specifies the "**" operator
- Parse command lines from the keyboard or specified files.

Before reading this chapter, you should be familiar with the user-visible features of CL. These features are described in detail in the DOMAIN System Command Reference manual and the DOMAIN System User's Guide.

CL calls are often used in conjunction with File & Tree Utility (FU) calls. For more examples of the CL, you might want to refer to Chapter 2, Using the File and Tree Utility (FU).

1.1. System Calls, Insert Files, and Data Types

To use the CL manager, use the system calls with the CL prefix. This chapter describes how most of these calls work. For details on CL call syntax, data types, and error messages, see Part II of this manual.

When using CL system calls in your program, you must specify the appropriate insert file for the language your are using. The CL insert files are:

- /SYS/INS/CL.INS.C for C.
- /SYS/INS/CL.INS.FTN for FORTRAN.
- /SYS/INS/CL.INS.PAS for Pascal.

1.2. Overview of the CL Manager

Before we describe how the CL manager works, we need to define a few terms that it uses.

A **token** is a text string that CL reads from the command line and parses according to the Shell parsing rules.

A **names-file** is a text file containing pathnames. CL reads the contents of a names-file as if it appeared on the command line. Tokens in a names-file can be delimited by spaces, or the NEWLINE character.

A **flag** is a token beginning with the hyphen, "-". Flags allow you to specify special actions so that the user can make optional choices on the command line.
While we refer to hyphenated words as flags, the DOMAIN System Command Reference manual refers to them as options. Also, some CL system calls refer to flags as keywords.

An argument is any token that is not a flag.

A wildcard-name is an argument that represents one or more existing pathnames. CL accepts wildcard characters as part of pathname arguments, and expands them to pathnames. These wildcard characters are listed in the DOMAIN System Command Reference manual.

A derived name is a name that is derived from another name. For example, Shell commands handle derived names with the = wildcard. The Shell command line $ cpf name =.derived copies the file *name* to *name.derived.* For details on derived names, see the DOMAIN System Command Reference manual.

When the CL manager reads the command line, it places each token from the command line and any names-files into an individual token record. The token records form a linked list or token list, which resides in scratch space. Initially, the token pointer points to the head of the list. When using CL system calls to read tokens from the list, the token pointer keeps track of the most recent token read.

Figure 1-1 shows the structure of a token list.

Command line:  command argument argument -flag *names-file

Token List:

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Argument</th>
<th>-Flag</th>
<th>Name from names-file</th>
<th>Name from names-file</th>
</tr>
</thead>
</table>

Token Pointer

Figure 1-1. Structure of a Command Line Token List

Once you initialize the CL, you make other CL calls to read the flags and names from the command line. When making these calls, the CL refers to the token list, not the command line or names-files.

NOTE: CL does not expand wildcards at initialization because it cannot determine which tokens represent names. Wildcard names and derived names are expanded later.

When you make calls to get tokens, CL checks the token list for the token and marks it "used." Usually, once CL marks a token used, you cannot refer to it again. You can refer to a used token when handling derived names, or using any of the CL_$REREAD system calls. See Part II of this manual for details.

When using the CL manager to parse your command line, you usually perform the following steps:
1. Use CL_$INIT to initialize the token list on which all subsequent CL calls operate.

2. Use the appropriate GET_FLAG system call to read the token list for flags.

3. If flags require arguments, get the associated arguments using the appropriate call: CL_$GET_NAME for pathnames, CL_$GET_NUM for numbers, or CL_$GETARGS for character strings.

4. Check the token list for unclaimed flags with CL_$CHECK_UNCLAIMED.

5. Establish a loop to read the token list for arguments using the appropriate system call: CL_$GET_NAME for pathnames, CL_$GET_NUM for numbers, or CL_$GETARGS for character strings.

Figure 1-2 shows the basic model of for parsing the command line using CL calls. The next few sections describe these calls in detail.

```
BEGIN { Main } 
    cl_$init ( . . . );
    { Get specified flags. } 
        IF cl_$get_flag ( . . . ) THEN . . . ; 
        { Get argument associated with flag. } 
        IF cl_$get_flag ( . . . ) THEN BEGIN 
            IF NOT cl_$get_arg ( . . . )  
                THEN error_routine  
            ELSE { Convert to desired type, if necessary.  
                        Handle arguments. } 
        END; { if } 
    { Check for any erroneous options that the user specified. } 
        cl_$check_unclaimed; 
    { Establish a loop to read names or other arguments. } 
        WHILE cl_$get_name ( . . . ) 
            DO BEGIN 
                { Do work here. } 
            END; { do begin } 
END. { Program }
```

Figure 1-2. Model for Parsing the Command Line
If an error occurs during a CL system call, CL prints an error message and performs a PGM_$EXIT and terminates your program. To prevent your program from terminating, you can set up a clean-up handler, as described in the Programming with General System Calls manual.

The next sections describe how to use specific CL calls to initialize the CL manager and get tokens from the token list.

1.3. Initializing the CL Manager

When using the CL manager to handle the command line, you must first initialize it by calling CL_$INIT or CL_$SETUP. CL_$INIT initializes CL and parses the command line according to the Shell parsing rules.

CL_$SETUP is like CL_$INIT in that it initializes CL manager, but it doesn't parse the command line. It allows you to supply a different source for the tokens, for example, from a file or interactively from the keyboard. You supply the additional CL parsing routines to perform the parsing. (For details, see Section 1.3.3.)

When you initialize CL, you can specify certain options to control how CL reads the tokens from the command line. These options affect how CL interacts with the user. For example, you can tell CL not to expand any wildcards, or to verify each pathname it reads with the user.

Table 1-1 lists the various CL calls that you can use to initialize CL and set up options. Sections 1.3.1 and 1.3.2 describe the options you can specify when initializing the CL manager. Section 1.3.3 is an example of initializing CL.

<table>
<thead>
<tr>
<th>System Call Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$INIT</td>
<td>Initializes the command line handler, parsing the command line according to the Shell parsing rules.</td>
</tr>
<tr>
<td>CL_$SETUP</td>
<td>Initializes the command line handler, but does not load anything to parse. Use this call when the tokens come from a file or keyboard instead of the command line. You call one of the following CL parse calls to specify where the tokens are.</td>
</tr>
<tr>
<td>CL_$PARSE_LINE</td>
<td>Supplies the line for CL to parse. On subsequent calls, CL returns information from this line. It discards any previous arguments.</td>
</tr>
<tr>
<td>CL_$PARSE_INPUT</td>
<td>Reads a line from the specified stream ID and hands it to CL_$PARSE_LINE. Returns TRUE if successful, FALSE if it encounters the end-of-file character.</td>
</tr>
</tbody>
</table>
Table 1-1. System Calls to Initialize the CL Manager, Cont.

<table>
<thead>
<tr>
<th>System Call Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL__PARSE_ARGS</td>
<td>Gets arguments from the keyboard. Use this after getting arguments with CL__GET_ARG. This call disregards the first argument, because it assumes that it is the command name.</td>
</tr>
<tr>
<td>CL__SET_WILD_OPTIONS</td>
<td>Defines the wildcard options for the wildcard manager.</td>
</tr>
<tr>
<td>CL__SET_NAME_PREFIX</td>
<td>Defines a character string prefix to add before each name specified on the command line. For example, you might want to add the prefix /sys/print, to names on a print queue command.</td>
</tr>
<tr>
<td>CL__SET_DERIVED_COUNT</td>
<td>Specifies the number of derived names allowed to follow each wildcard name. The default value is 1.</td>
</tr>
<tr>
<td>CL__SET_OPTIONS</td>
<td>Adds specified options to the CL option set defined in the CL__INIT or CL__SETUP call.</td>
</tr>
<tr>
<td>CL__RESET_OPTIONS</td>
<td>Replaces the previously defined CL option set with the specified CL option set.</td>
</tr>
</tbody>
</table>

1.3.1. Defining CL Options

You can control how the CL reads the command line by specifying CL options with the CL\_\_INIT or CL\_\_SETUP call. You can let CL define a set of these options automatically, by specifying the default empty brackets, []). Or, you can specify alternative CL options, by listing the CL options in the brackets.

Table 1-2 lists the default set of options, and their alternatives. Note that you can specify only one alternative per default option; the alternatives are mutually exclusive.
## Table 1-2. CL Options

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CL_$WILDCARDS</strong></td>
<td><strong>CL_$NO_WILDCARDS</strong></td>
</tr>
<tr>
<td>Causes CL_$GET_NAME to expand</td>
<td>Causes CL_$GET_NAME to return wildcard-names verbatim; it does</td>
</tr>
<tr>
<td>wildcards and return the expanded</td>
<td>not expand them.</td>
</tr>
<tr>
<td>names.</td>
<td></td>
</tr>
<tr>
<td><strong>CL_$NO_MATCH_WARNING</strong></td>
<td><strong>CL_$NO_MATCH_OK</strong></td>
</tr>
<tr>
<td>Displays a warning message on error</td>
<td>Displays no warning or error message when a wildcard does not match</td>
</tr>
<tr>
<td>output when a wildcard does not match</td>
<td>existing pathnames.</td>
</tr>
<tr>
<td>existing pathnames.</td>
<td></td>
</tr>
<tr>
<td><strong>CL_$VERIFY_NONE</strong></td>
<td><strong>CL_$VERIFY_WILD</strong></td>
</tr>
<tr>
<td>CL_$GET_NAME does not verify any</td>
<td>CL_$GET_NAME verifies only names expanded wildcards with the</td>
</tr>
<tr>
<td>names with the by user.</td>
<td>user.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CL_$STAR_NAMES</strong></td>
<td><strong>CL_$NO_STAR_NAMES</strong></td>
</tr>
<tr>
<td>Allows the user to specify names-files</td>
<td>Does not allow users to specify names-files. Treats the ** operator.</td>
</tr>
<tr>
<td>with the ** operator.</td>
<td></td>
</tr>
<tr>
<td><strong>CL_$KEYWORD_DELIM</strong></td>
<td><strong>CL_$NO_KEYWORD_DELIM</strong></td>
</tr>
<tr>
<td>Prevents CL_$GET_ARG and CL_$GET_</td>
<td>Causes no special treatment of unread flags by CL_$GET_ARG and</td>
</tr>
<tr>
<td>NAME from returning an &quot;unused&quot; flag</td>
<td>CL_$GET_NAME. The procedures return the flags as if they were</td>
</tr>
<tr>
<td>as an argument. The calls return FALSE</td>
<td>arguments.</td>
</tr>
<tr>
<td>if they find a flag.</td>
<td></td>
</tr>
<tr>
<td><strong>CL_$DASH_NOP</strong></td>
<td><strong>CL_$DASH_NAMES</strong></td>
</tr>
<tr>
<td>CL treats the hyphen &quot;-&quot; as a name.</td>
<td>CL reads names from standard input when it finds a hyphen.*</td>
</tr>
<tr>
<td>Normally, the hyphen is an identifier</td>
<td></td>
</tr>
<tr>
<td>for the standard input stream.</td>
<td></td>
</tr>
</tbody>
</table>
Table 1-2. CL Options, Cont.

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$DASH_DFT_NOP</td>
<td>Suppresses any special action when there are no arguments on the command line.</td>
</tr>
<tr>
<td>CL_$NO_COMMENTS</td>
<td>Causes no special treatment of characters enclosed in brackets.</td>
</tr>
<tr>
<td></td>
<td>CL_$NAME_DFT_STDIN</td>
</tr>
<tr>
<td></td>
<td>Causes CL to read names from standard input if no arguments appear on the command line.*</td>
</tr>
<tr>
<td></td>
<td>CL_$COMMENTS</td>
</tr>
<tr>
<td></td>
<td>Causes CL to ignore all characters enclosed in brackets when reading a names-file. Do not use this in commands that accept derived names because you must use brackets to specify the tag expressions you want to use in a derived name.</td>
</tr>
</tbody>
</table>

* Made available to remain compatible with previous software releases. Obsolete for new software development.

You can add options to the current CL option set at another time during your program by using the CL_$SET_OPTIONS call. If you want to remove options from the current CL option set, you can specify an entirely new set of CL options with CL_$RESET_OPTIONS. CL_$RESET_OPTIONS replaces the previously defined CL option set with the option set specified in the call.

Example 1-1 shows how a program changes the set of CL options.

**Initialize CL with specified CL options:**

This CL options set contains CL_$NO_WILDCARDS, plus the default options: CL_$VERIFY_NONE, CL_$STAR_NAMES, CL_$KEYWORD_DELIM, CL_$DASH_NOP, CL_$DASH_DFT_NOP, CL_$NO_COMMENTS.

```c
cl_$init ([cl_$no_wildcards], {Returns wildcards literally})
    program_name, {Name of program}
    sizeof(program_name)); {Length of program name}
```

**Add to the current set of CL options:**

This CL options set adds CL_$VERIFY_ALL to the option set listed above:

```c
cl_$set_options ([cl_$verify_all]); {Verify all names}
```

Example 1-1. Redefining the CL Options Set
Replace the current list of options with a new set:

This CL options set contains the default set of options: CL_$_WILDCARDS, CL_$_VERIFY_NONE, CL_$_STAR_NAMES, CL_$_KEYWORD_DELIM, CL_$_DASH_NOP, CL_$_DASH_DFT_NOP, CL_$_NO_COMMENTS.

Example 1-1. Redefining the CL Options Set, Cont.

1.3.2. Defining Wildcard Options

When you initialize the CL manager, it defines a default set of wildcard options that determine how the wildcard manager expands wildcards. You can change how the CL wildcard manager expands wildcards by specifying other wildcard options with the CL_$_SET_WILD_OPTIONS system call.

Table 1-3 lists the wildcard options available. You can list any combination of wildcard options with CL_$_SET_WILD_OPTIONS. Note that you must specify all the wildcard options you want in effect, including the default options.

<table>
<thead>
<tr>
<th>Default?</th>
<th>Wildcard Option</th>
<th>Wildcards match:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>CL$_WILD_FILES</td>
<td>Names of files.</td>
</tr>
<tr>
<td>Yes</td>
<td>CL$_WILD_DIRS</td>
<td>Names of directories.</td>
</tr>
<tr>
<td>Yes</td>
<td>CL$_WILD_LINKS</td>
<td>Names of links.</td>
</tr>
<tr>
<td>No</td>
<td>CL$_WILD_EXCLUSIVE</td>
<td>Only the highest directory of a given wildcard. This is useful for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>commands that operate on entire directories such as COPY_TREE. Since it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operates on all subdirectories, there's no need to match further.</td>
</tr>
<tr>
<td>No</td>
<td>CL$_WILDCHASE_LINKS</td>
<td>Files and directories pointed to by links.</td>
</tr>
<tr>
<td>No</td>
<td>CL$_WILD_FIRST</td>
<td>Only the first name of a given wildcard rather than expanding all names.</td>
</tr>
</tbody>
</table>

1.3.3. Example of Initializing the CL Manager

Normally, you initialize the CL manager with CL$_INIT so that CL can parse the command line automatically. (See Example 1.7 for a sample program using CL$_INIT.) Examples in this section show how to initialize the CL manager with CL$_SETUP, and supply the lines to parse.
Example 1-2 is a program segment using CL_$SETUP to initialize the CL, and CL_$PARSE_LINE to supply the arguments for parsing. Instead of reading the command line for arguments, this program asks the user to supply a filename that contains the arguments to parse. It gets the file with STREAM calls, and then parses each line in the file with CL_$PARSE_LINE.

```pascal
program cl_parse_line;

%no_list;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/cl.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';
%list;

CONST
  program_name = 'cl_parse_line';

VAR
  pathname : name_$pname_t;
  name_len : integer;
  buf : name_$pname_t;
  ret_ptr : 'name_$pname_t';
  ret_len : integer32;
  seek_key : stream$_sk_t;
  stream_id : stream$_id_t;
  status : status$_t;

PROCEDURE check_status; { Error handling procedure }{
BEGIN { Main }
{ Initialize CL with the default options, but don't parse the command line. }
  cl_$setup([], program_name, sizeof(program_name));

{ Set wildcard options. Note, if you want more than the default wildcard options, you must specify all the wildcard options that you want. }
  cl_$set_wild_options ([cl_$wild_files,
                          cl_$wild_dirs,
                          cl_$wild_links,
                          cl_$wild_first]);

{ Ask user for the file containing the lines. }
  writeln ('Enter the file containing the command line arguments.');

Example 1-2. Initializing CL with CL_$SETUP and CL_$PARSE_LINE
```
vfmt_read2 ( '%e%ex%ek%', count, status, pathname, name_len);
check_status;

{ Open file to parse and get lines. }
stream_open ( pathname, name_len, stream_read, stream_no_conc_write,
             stream_id, status);
check_status;

WHILE (status.all = status_ok) DO BEGIN { Read lines until end-of-file. }

stream_get_rec( stream_id, addr(buf), sizeof(buf), ret_ptr,
                ret_len, seek_key, status );

IF (status.code = stream_end_of_file) AND
   (status.subsys = stream_subs) THEN EXIT;
check_status;

{ Parse the line before the NEWLINE character. }
cl_parse_line( ret_ptr, ret_len -1 );

END; { while }

stream_close ( stream_id, status);
check_status;

Example 1-2. Initializing CL with CL_$SETUP and CL_$PARSE_LINE, Cont.

While CL_$PARSE_LINE is useful for getting command line arguments from a file,
CL_$PARSE_INPUT is useful for getting command line arguments interactively. The
advantage of using CL_$PARSE_INPUT over CL_$PARSE_LINE is that it gets the line to
parse and parses it in one step. (CL_$PARSE_LINE requires that you supply the line to
parse.)

Example 1-3 is an example of initializing the CL with CL_$SETUP and CL_$PARSE_INPUT.

program cl_parse_input;
%olist:
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/cl.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';
%list;

CONST
  program_name = 'cl_parse_input';

Example 1-3. Initialising CL with CL_$SETUP and CL_$PARSE_INPUT
Example 1-3. Initializing CL with CL\_$SETUP and CL\_$PARSE\_INPUT, Cont.

1.4. Getting Flags

After initializing the CL manager, you usually check for any flags the user might have specified. The three CL calls that check for flags are the following:

**CL\_$GET\_FLAG**
Checks for a specified flag, marks it used, and counts the number of tokens following it, up to the next flag or the end of the list.

**CL\_$CHECK\_FLAG**
Checks for a specified flag, marks it used, and checks for correct number of tokens following it.

**CL\_$GET\_ENUM\_FLAG**
Scans the token list for one of several flags the caller might have specified. It counts the number of tokens following it, up to the next flag or the end of the list.

Your program can accept an abbreviated or full version of any flag by using brackets. For example, the string "-brief" allows the user to specify either "-br" or "-brief." CL will return an error if the user specifies "-brie."
NOTE: You must use lowercase letters when defining flags with a GET_FLAG system call. CL will not recognize uppercase letters. However, the user can type either uppercase or lowercase letters.

Another way to get arguments associated with a flag is to specify CL_$NEXT as the first argument in the CL_$GET_ARGS or CL_$GET_NAME system calls. See Section 1.5 for details.

CL_$GET_FLAG and CL_$GET_ENUM_FLAG returns the number of tokens following the flag, up to the next flag. Note that these calls count the number of tokens on the command line only. The count does NOT include names expanded from wildcards.

Normally, you use the GET_FLAG calls to get the flags from the token list before getting the names. In this case, after you make the calls to get the flags, you call CL_$CHECK_UNCLAIMED. CL_$CHECK_UNCLAIMED checks for any invalid flags the user might have specified. CL prints an error message on error output and aborts the program if it finds any unclaimed flags.

There is one case where you might not want to call CL_$CHECK_UNCLAIMED: When you use the CL_$GET_FLAGGED_DERIVED_NAME call to associate a flag with a particular name, you cannot call CL_$CHECK_UNCLAIMED before getting the names. For more information on this call, see Section 1.5.

Sections 1.4.1 and 1.5.1 include Pascal examples of getting flags. Section 1.4.1 shows how to use the CL_$GET_ENUM_FLAG system call to define synonymous options that require the same program action. Section 1.5.1 shows how to get an argument associated with a specific flag. See Section 1.7 for a sample program using other GET_FLAG system calls.

1.4.1. Example of Handling Synonymous Flags with CL_$GET_ENUM_FLAG

Example 1-4 shows how to use the CL_$GET_ENUM_FLAG system call to define synonymous options that require the same program action. The CL_$GET_ENUM_FLAG allows you to specify an action if the user typed one of several flags. For example, the flags "-error" and "-fault" might both require the same action. Each time you call CL_$GET_ENUM_FLAG system call, CL looks for the specified flags on the token list. If it finds any flag specified, it returns the index of the flag in the list, and stops searching. If it does not find a specified flag, it returns a zero.

You can find out which flag the user actually typed in one of the following ways: You can manipulate the index, since the CL_$GET_ENUM_FLAG call returns the index of the flag found. Or you can use the CL_$GET_FLAG_INFO system call to get the exact text that the user wrote. This example uses the CL_$GET_FLAG_INFO system call to see what the user actually typed.
PROGRAM cl_get_enum_flag;
#include "sys/ins/cl.ins_pas";

VAR
  report : integer;
  number : integer;
  flag_ptr : "string;
  flag_len : integer;

{ Report message if the user types the -message or -report flags. }
report := cl_$get_enum_flag ( cl$_first, "-message -report X", number );
{ Check for any undefined flags that the user might have typed. }
cl$get_unclaimed;

IF report <> 0 THEN BEGIN
  writeln ( 'This is the message you requested.' );
  cl$_get_flag_info (flag_ptr,
                    flag_len);
  writeln ( 'You typed the option: . flag_ptr" : flag_len);
END;

Example 1-4. Using CLGNUC_ENUM_FLAG

1.5. Reading Arguments

After you get the flags to handle the options that the user specified, you get the remaining arguments on the token list. To do so, call one of the following system calls:

CLGNUC_NUM
  Gets the next unused integer from the token list; converts it from a decimal to a 4-byte integer.

CLGNUC_NAME
  Gets the first or next unused name from the token list. It expands wildcards.

CLGNUC_ARG
  Gets the first or next unused argument from the token list, in character string format.
To get the arguments, you normally use the system call within a loop. This way, CL will continue to scan through the token list and return arguments until there are none left.

Where the CL_GET calls begin the search depends on whether you set the first argument of the call to CL_$FIRST or CL_$NEXT. If you specify CL_$FIRST, CL begins searching from the head of the list each time it scans the token list. If you specify CL_$NEXT, it begins the search at the token pointer, which is set to the last token read. Normally, you use CL_$NEXT when you want the argument adjacent to the last token read.

Sections 1.5.1 and 1.5.2 demonstrate ways to get arguments from the command line.

1.5.1. Getting Arguments Associated with Flags

You can get arguments associated with flags by using a either the CL_$GET_ARG, CL_$GET_NAME, or CL_$GET_NUM system call.

Use CL_$GET_NAME if you expect a pathname, and CL_$GET_NUM if you expect a decimal number. If you want to accept both names or numbers, you can use CL_$GET_ARG. Since CL_$GET_ARG returns arguments in character string format, it will accept any argument. You can then convert the argument to the desired type. (The system call VFMT_$DECODE2 allows you to convert variables to another type.)

Example 1-5 uses a CL_$GET_ARG, specifying CL_$NEXT as the first argument, so that CL_$GET_ARG will get the next argument after the "-copies" flag. Then it uses a VFMT_$DECODE system call to convert the string into a number. Note that we could have used CL_$GET_NUM to get the number directly, but CL_$GET_ARG assures that we get the token immediately following the flag.
PROGRAM cl_get_args;

#include '/sys/ins/cl.ins.pas';

VAR
  num_args : integer;
  copies_string : string;
  copies_len : integer;
  copies_num : integer;
  hunoz : integer;
  huares : integer;

BEGIN { Main }

  IF cl_$get_flag ('-copies', num_args) THEN
    BEGIN
      IF NOT cl_$get_arg (cl_$next, copies_string, copies_len, sizeof(copies_string)) THEN BEGIN
        writeln (' You must specify a number after "-copies."');
        pgm_$set_severity (pgm_$error);
        pgm_$exit
      END;

      { Convert copies_string to a number. }
      hunoz := vfmt_$decode2 ('%wd', copies_string, copies_len, huares, status, copies_num, 0);
    END;

END;

Example 1-5. Getting Arguments Associated with Flags

1.5.2. Getting Derived Names

If your program expects derived names, you can retrieve them using one of the following GET_DERIVED_NAME calls:

CL_$GET_DERIVED_NAME
  Gets the next derived name associated with the last name returned from CL_$GET_NAME.

CL_$GET_FLAGGED_DERIVED_NAME
  Gets the next derived name associated with the specified flag, and the last name returned from CL_$GET_NAME.
The following example shows how to use the CL_$GET_NAME, CL_$GET_DERIVED_NAME, and CL_$GET_FLAGGED_DERIVED_NAME system calls. This program expects two arguments, the second name is a derived name from the first argument. If the user supplied the "-list" flag, it prints out the second name.

```pascal
PROGRAM cl_get_flagged_derived_names;

%include '/sys/ins/cl.ins.pas';

VAR
  select : cl_arg_select_t;
  count : integer;
  name : array [1..2] of name_pname_t;
  len : array [1..2] of integer;
  num_args : integer;

BEGIN { Main }
  { Initialize the CL manager }
  { Get flags except '-list' }

  { Get first argument. }
  select := cl_first;
  WHILE cl_get_name ( select,
    name [1],
    len [1],
    sizeof(name [1]) )
  DO BEGIN
    { Get second argument. Check to see if it is a derived name. If it is, change name to derived name. }
    select := cl_next;
    IF cl_get_derived_name (name [2],
      len [2],
      sizeof(name [2]))
     THEN { Use FU manager to copy file }
     BEGIN
      IF cl_get_flagged_derived_name ('-list',
        cl_required,
        name[2],
        len[2],
        sizeof(name[2]))
      THEN BEGIN
        IF len[2] = 0 THEN { Command line error }
          writeln ('You can supply only one derived name per flag. ')
        ELSE { Write name }
          vfmt_write2 ( '%a',
            name[2],
            len[2] );
        END; { begin }
      END; { begin }
    END; { begin }
  END; { begin }
```

Example 1-6. Reading Arguments from the Token List
1.6. Using Miscellaneous CL Calls

The following is a list of miscellaneous CL calls that may be useful. The sample program in Section 1.7, as well as other examples in this chapter, shows how many of these calls work.

**Table 1-4. Miscellaneous CL System Calls**

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$VERIFY</td>
<td>Verifies with the user that the program is supposed to operate on this name.</td>
</tr>
<tr>
<td>CL_$SET_VERB</td>
<td>Defines a verb for CL to display before each pathname when querying the user for names.</td>
</tr>
<tr>
<td>CL_$GET_FLAG_INFO</td>
<td>Returns the actual text of the previously returned flag.</td>
</tr>
<tr>
<td>CL_$GET_NAME_INFO</td>
<td>Returns information about the previously returned name. Determines whether the user supplied a full pathname or wildcard name on the command line.</td>
</tr>
<tr>
<td>CL_$MATCH</td>
<td>Compares a token against a specified string. Useful for cases where the command line logic is too complicated for CL to handle.</td>
</tr>
<tr>
<td>CL_$GET_SET</td>
<td>Creates a set to compare two character strings.</td>
</tr>
<tr>
<td>CL_$SET_STREAMS</td>
<td>Tells CL to use the specified streams as the default input and output channels.</td>
</tr>
<tr>
<td>CL_$REREAD</td>
<td>Marks the entire token list &quot;unused&quot; so the flags, names and arguments can be read again.</td>
</tr>
<tr>
<td>CL_$REREAD_NAMES</td>
<td>Marks all names &quot;unused&quot; so the names can be read again. The next CL_$GET_NAME call returns the first name on the list.</td>
</tr>
<tr>
<td>CL_$REREAD_FLAGS</td>
<td>Marks all the flags &quot;unused&quot; so the flags in the token list can be read again.</td>
</tr>
</tbody>
</table>
1.7. Sample Program Using CL System Calls

Example 1-7 uses CL system calls to parse the command line. It also uses File and Tree Utility (FU) system calls to copy a file. This program checks to see if the user specified a derived name, and copies the file to the new extension. For more information on FU system calls, see Chapter 2, Using the File and Tree Utility (FU).

```pascal
PROGRAM cl_copy_file;

%inolist;
%include '/sys/ins/ubase.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/cl.ins.pas';
%include '/sys/ins/fu.ins.pas';
%list;

CONST
  name_max = 32;
  program_name = 'copy_derived_names';

VAR
  select : cl$_arg_select_t;
  count : integer;
  name : array [1..2] of name$ pname_t;
  len : array [1..2] of integer;
  no_args : integer;
  status : status_t;

  fu_opts : fu$opt_set_t;
  fu_context : fu$scontext_t;

{ Internal Procedure -- Print_error ================ }

PROCEDURE print_error;
BEGIN
  pgm$set_severity (pgm$error);
  pgm$exit;
END;

{ ******************************************************* }

BEGIN { Main }

{ Initialize CL with the default set of CL options. }

  cl$init ( [] ,
    program_name ,
    sizeof (program_name) );

  fu$init;
```

Example 1-7. Sample Program Using CL System Calls
fu_$set_prog_name ( program_name, sizeof (program_name) );

fu_opts := ([ fu$coe,
    fu_print_errors,
    fu$_dacl,
    fu$_subs ]);:

{ Set verb for query in case user uses wildcards. }
c1$_set_verb ('Copy', 4 ); { Verb. length of verb }

{ Allow the user to specify only 1 derived name after each wildcard name. }
c1$_set_derived_count (1);

{ Set the verify option that the user specifies. }

IF c1$_get_flag ('-qa', no_args) THEN { Query all names }
c1$_set_options ([c1$_verify_all]);
IF c1$_get_flag ('-qw', no_args) THEN { Query names expanded by wildcards }
c1$_set_options ([c1$_verify_wild]);
IF c1$_get_flag ('-nq', no_args) THEN { No Query }
c1$_set_options ([c1$_verify_none]);

{ Check for any invalid flags; exit if the user specified any. }
c1$_check_unclaimed;

{ Get first name from the token list. }
select := c1$_first:
WHILE c1$_get_name ( select,
    name [1],
    len [1],
    sizeof (name [1]))
    DO BEGIN
        { Get second name. Copy the first name to the
         second name. }
        select := c1$_next;
        IF c1$_get_derived_name ( name [2],
            len [2],
            sizeof (name [2]))
            THEN BEGIN
                fu$_copy_file ( name [1],
                    len [1],
                    name [2],
                    len [2],
                    fu_opts,
                    fu_context,
                    status );
                IF status.all <> 0 THEN
                    BEGIN
                        print_error; { FU error message }
                    END;
                writeln ( name[2] :len[2], ' created.' ); { Confirm }
            END
        END
    END. { Program }

Example 1-7. Sample Program Using CL System Calls, Cont.
Output

The following is a sample output from the above program.

```
$ cl_copy_file.bin existing_file create_existing_file
    create_existing_file created.
$ cl_copy_file.bin no_file no_file_2
    *(copy_derived_names) "no_file" - name not found (OS/naming server)
$ cl_copy_file.bin test-derived =.chek
    test-derived.chek created.
$ cl_copy_file.bin test/** =.all -qa
    Verify wildcard "test/**":
    Copy "test2" ? y
    *(file) "test2.all" deleted.  *(= Replaced contents of existing file. *)
    TEST2.all created.
    Copy "test2.all" ? n  *(= Queries file just created. *)
    Copy "test_fu.chek" ? y
    TEST_FU.CHEK.all created.
    Copy "test_tree" ? y
    *(copy_derived_names) "test_tree" - object must be a leaf (US/file utility)
```

Example 1-7. Sample Program Using CL System Calls, Cont.
Chapter 2
Using the File and Tree Utility

The File and Tree Utility (FU) is a set of DOMAIN system routines that provides a consistent way to handle files and directories. By using FU system calls, your program can handle certain operations in the same way the Shell handles them so that you can provide your users with a consistent interface. FU allows you to perform the following operations:

- Copy files.
- Copy, merge, and replace files and directories.
- Delete files and directories.
- Compare directories.
- Move files.
- Make a unique pathname by appending today's date.

Most of these system calls correspond to individual Shell commands. For more information on these commands, see the DOMAIN System Command Reference manual.

This chapter provides an overview on using the FU subsystem and a sample program in Pascal. It concludes with system call syntax, data type, and error information.

Since FU calls are often used in conjunction the Command Line Handler (CL), you might want to see Chapter 1, Parsing the Command Line for more information on the CL.

2.1. System Calls, Insert Files, and Data Types

To use the File and Tree Utility, use the system calls with the FU prefix. This chapter describes how most of these calls work. For details on FU call syntax, data types, and error messages, see Part II of this manual.

When using FU system calls in your program, you must specify the appropriate insert file for the language you are using. The FU insert files are

(SYS/INS/FU.INS.C) for C.
(SYS/INS/FU.INS.FTN) for FORTRAN.
(SYS/INS/FU.INS.PAS) for Pascal.

2.2. Overview of the FU System Calls

The File and Tree Utility (FU) allows you to perform common operations on files and trees using DOMAIN system routines. When using FU, you often use the calls in the following order:
1. Initialize FU with FU_$INIT to allocate temporary storage space required to perform subsequent FU operations.

2. Identify your program name with FU_$SET_PROG_NAME so that FU can identify your program by name when reporting errors.

3. Use the various calls to operate on files and trees.

4. Release the storage space used by FU with FU_$RELEASE_STORAGE.

Table 2-1 lists the operations you can perform on files and trees. It also lists each call's corresponding Shell command.

<table>
<thead>
<tr>
<th>System Call</th>
<th>Corresponding Shell Command</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU_$CMP_TREE</td>
<td>CMT</td>
<td>Compares a source tree to a target tree.</td>
</tr>
<tr>
<td>FU_$COPY_FILE</td>
<td>CPF</td>
<td>Copies a file from the source pathname to the target pathname.</td>
</tr>
<tr>
<td>FU_$COPY_TREE</td>
<td>CPT</td>
<td>Copies, merges, and replaces files, directories, and links.</td>
</tr>
<tr>
<td>FU_$DELETE_FILE</td>
<td>DLF</td>
<td>Deletes a specified file.</td>
</tr>
<tr>
<td>FU_$DELETE_TREE</td>
<td>DLT</td>
<td>Deletes a tree and all its descendants.</td>
</tr>
<tr>
<td>FU_$MOVE_FILE</td>
<td>MVF</td>
<td>Moves a file to a different location in the naming tree.</td>
</tr>
<tr>
<td>FU_$RENAME_UNIQUE</td>
<td>CHN-U</td>
<td>Renames a pathname to create a unique name by appending today's date.</td>
</tr>
</tbody>
</table>

The FU system calls require that you specify the source and, where applicable, target pathnames and pathname lengths. In addition, most calls require that you set FU options. The options control how FU performs the operation. For example, you can set FU options to list files as they are operated, or to set the ACL of the target files.

2.2.1. Setting FU Options

By setting FU options, you can provide users with options similar to those on the standard Shell commands. To do so, use the CL_$GET_FLAG system call to check for the option; then set FU options by assigning the appropriate predefined values to the FU_OPTIONS variable. (See Example 2-1.)

Table 2-2 lists the predefined FU options and their corresponding Shell options.
Table 2-2. FU Options To Provide Shell Command Options

<table>
<thead>
<tr>
<th>FU Option</th>
<th>Corresponding Shell Option</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU_$AFT_TIME</td>
<td>-AF DATE</td>
<td>Operates on only those objects whose dtm (date/time last modified) is after the given date and time.</td>
</tr>
<tr>
<td>FU_$BEF_TIME</td>
<td>-BE DATE</td>
<td>Operates on only those objects whose dtm (date/time last modified) is before the given date and time.</td>
</tr>
<tr>
<td>FU_$COE</td>
<td>Not specifying -AE (Abort on error)</td>
<td>If an error occurs while processing a file, continues to the next file. Most programs set this option.</td>
</tr>
<tr>
<td>FU_$DAACL</td>
<td>-DAACL</td>
<td>Assigns default ACL to target files. Target gets the same ACL as the parent (destination) directory.</td>
</tr>
<tr>
<td>FU_$DEL_WHEN_UNLKD</td>
<td>-DU</td>
<td>Deletes object when it becomes unlocked.</td>
</tr>
<tr>
<td>FU_$FORCE</td>
<td>None</td>
<td>Forces a copy in FU_$MOVE if the source and target files are not located on the same volume.</td>
</tr>
<tr>
<td>FU_$FORCE_DEL</td>
<td>-F</td>
<td>Forces deletion of a target during a replace operation if user has protect (<em>P</em>) rights.</td>
</tr>
<tr>
<td>FU_$HELP</td>
<td>None</td>
<td>Displays detailed usage information. (This has no affect under AEGIS, but was added to provide help in a Boot Shell utility.)</td>
</tr>
<tr>
<td>FU_$LIST_D_DIRS</td>
<td>-LD</td>
<td>Lists directories operated on.</td>
</tr>
<tr>
<td>FU_$LIST_FILES</td>
<td>-LF</td>
<td>Lists files operated on.</td>
</tr>
<tr>
<td>FU_$LIST_LINKS</td>
<td>-LL</td>
<td>Lists links operated on.</td>
</tr>
<tr>
<td>FU_$LIST_D_DIRS</td>
<td>-LDL</td>
<td>Lists directories deleted as a result of a replace operation.</td>
</tr>
<tr>
<td>FU Option</td>
<td>Corresponding Shell Option</td>
<td>Operation</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FU_$LIST_D_FILES</td>
<td>-LF</td>
<td>Lists files deleted as a result of a replace operation.</td>
</tr>
<tr>
<td>FU_$LIST_D_LINKS</td>
<td>-DLL -L</td>
<td>Lists links deleted as a result of a replace operation.</td>
</tr>
<tr>
<td>FU_$LIST_DEL</td>
<td>-LDL -L</td>
<td>Lists objects deleted as a result of a replace operation. This is obsolete for new development; use FU_$LIST_D_FILES, FU_$LIST_D_DIRS and FU_$LIST_D_LINKS instead.</td>
</tr>
<tr>
<td>FU_$MERGE</td>
<td>CPT -MS</td>
<td>Merges source and target if both are directories. For files and links with the same name in source and target, it deletes the target, and replaces it with a copy of the source.</td>
</tr>
<tr>
<td>FU_$MERGE_DST</td>
<td>CPT -MD</td>
<td>Merges source and target if both are directories. For files and links with the same name in source and target, the target remains unchanged.</td>
</tr>
<tr>
<td>FU_$PRESERVE_DT</td>
<td>-PDT</td>
<td>Preserves the source dtm (date/time last modified) and dtu (date/time last used).</td>
</tr>
<tr>
<td>FU_$PRINT_ERRORS</td>
<td>None</td>
<td>Prints errors on the error output stream. Use this to report FU errors.</td>
</tr>
<tr>
<td>FU_$QUIT</td>
<td>None</td>
<td>Terminates the program. (This has no affect under AEGIS, but was added to provide help in a Boot Shell utility.)</td>
</tr>
<tr>
<td>FU_$RENAME</td>
<td>-CHN</td>
<td>Changes the name of an existing object with the target pathname before creating a copy. If the target name exists, it appends today's date to the target pathname.</td>
</tr>
<tr>
<td>FU_$REPLACE</td>
<td>-R</td>
<td>Replaces target with a copy of the source.</td>
</tr>
<tr>
<td>FU_$SACL</td>
<td>-SACL</td>
<td>Assigns ACL of source file to target file. Target gets the same ACL as the source file.</td>
</tr>
<tr>
<td>FU_$SUBS</td>
<td>-SUBS</td>
<td>Retains the source ACL for objects that belong to protected subsystems.</td>
</tr>
</tbody>
</table>
Example 2-1 shows how to set FU options. After initializing FU, the program sets up a default set of FU options. It then uses the CL $GET FLAG system call to check for the options on the command line. If the user specified an option, it sets the appropriate FU option.

```

Example 2-1. Setting FU Options

Apollo Confidential 2-5  Handling Files and Trees with FU
```
2.2.2. Releasing Storage with FU_$RELEASE_STORAGE

FU allocates read/write (RWS) storage space when copying files during the FU_$COPY_FILE, FU_COPY_TREE and FU_MOVE_FILE routines. (It needs this space for tables to keep track of ACLs.)

This storage gets released automatically when your program terminates. However, if your program performs numerous copy operations, we recommend that you call FU_RELEASE_STORAGE to release the storage more often.

2.3. Sample Program Using the FU System Calls

Example 2-2 is a sample program that uses FU system calls to perform common operations on files and directories. It also uses Command Line Handler (CL) calls to parse the command line. For more information on CL system calls, see Chapter 1, Parsing the Command Line.

This program allows the user to perform various FU operations. It asks the user to specify which operation to perform, then it prompts the user for pathnames and options.

```
program fu_handling_files_trees;

#include '/sys/ins/base.ins.pas';
#include '/sys/ins/streams.ins.pas';
#include '/sys/ins/cl.ins.pas';
#include '/sys/ins/fu.ins.pas';
#include '/sys/ins/vfmt.ins.pas';
#include '/sys/ins/scratch.ins.pas';
#include '/sys/ins/error.ins.pas';
#include '/sys/ins/time.ins.pas';
#include '/sys/ins/cal.ins.pas';

CONST
  program_name = 'fu_handling_files_trees';

VAR
  select : cl_arg_select_t;
  pathname : name_pname_t;
  pathname_len : integer;
  num_args : integer;
  status : status_s;
  command : string;
  command_len : integer;
  ok : boolean;
  null_line : boolean;
  fu_opts : fu_opt_set_t;
  fu_context : fu_context_t;
  fu_error : name_pname_t;
  fu_error_len : integer;
```

Example 2-2. Operating on Files and Trees with FU Calls
{ Internal Procedure -- Check_status ---------------------------------- }

PROCEDURE check_status;
BEGIN

IF status.all <> status_ok THEN

BEGIN

   error_print (status);
   pgm_set_severity (pgm_error);
   pgm_exit;
END;
END; { check_status }

{ Internal Procedure -- Get_time ---------------------------------- }

PROCEDURE get_time ( OUT int_clock : time_clock_t );

VAR

   readable_dt : cal_timedate_rec_t;
   date_string : string;
   date_len : integer;
   time_string : string;
   time_len : integer;
BEGIN

   { Convert command line date string to readable format.  
     Date must be in year/month/day (85/09/15) format.  }

   ok := cl_get_arg ( cl_next, date_string, date_len, sizeof(date_string) );

   cal_decode_ascii_date ( date_string, date_len, readable_dt.year,   
      readable_dt.month, readable_dt.day, status );
   check_status;

   { Convert command line time string to readable format.  
     Time must be in hr/min/sec 24-hour format (17:33:55).  }

   ok := cl_get_arg ( cl_next, time_string, time_len, sizeof(time_string) );

   cal_decode_ascii_time ( time_string, time_len, readable_dt.hour,   
      readable_dt.minute, readable_dt.second, status );
   check_status;

   { Convert readable date and time to internal time.  }

   cal_encode_time ( readable_dt, int_clock );
   cal_remove_local_offset ( int_clock );

END; { get_time }

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
{ Internal Procedure -- Copy_file }  

{ Copies source pathname to target pathname. By default, the target gets  
the ACL of the destination directory, unless user specifies -SACL.  
User can also preserve source's dtm/dtu times by typing -PDT. }  

PROCEDURE copy_file;  
VAR  
    new_pathname : name_$pname_t;  
    new_pathname_len : integer;  
BEGIN  

{ Set wildcard options. }  
    cl_$set_wild_options ( [cl_$wild_files] );  

{ Reinitialize default FU options. }  
    fu_opts := (
        [fu$_coe,  { Continue on error }  
        fu$_print_errors,  { Print FU errors }  
        fu$_dacl,  { Target gets ACL of destination directory }  
        fu$_subs ]);  { Retain source ACL for objects in subsytems }  

{ Get the filenames. }  
    writeln;  
    writeln ( ' Enter the name of the file you want to copy, then ' );  
    writeln ( ' the name of the new file you want created. ' );  
    writeln;  
    writeln;  
    writeln ( ' Type "-r[replace]" if you want to replace the target with ' );  
    writeln ( ' the source. If target exists, it will be deleted. ' );  
    writeln ( ' If it doesn'  ' t exist, it will be created. ' );  
    writeln;  
    writeln ( ' Type "-sacl" if you want the target file to have the ' );  
    writeln ( ' same ACL as the source file; otherwise the target ' );  
    writeln ( ' file will have the same ACL as its parent directory. ' );  
    writeln;  
    writeln ( ' Type "-pdt" if you want to preserve the source file' ' s ' );  
    writeln ( ' modification and used times. ' );  
    writeln;  

    ok := cl_$parse_input ( stream$_stdin, null_line );  

IF null_line THEN  
    writeln ( 'No input. Type CTRL/Q to quit.' );  
  
{ Get keywords, set appropriate FU options. Check for any invalid keywords; 
and exit if user specified any. }  

IF cl_$get_flag ( '-r[replace]', num_args ) THEN  
    fu_opts := fu_opts + [fu$_replace];  { Replace target with source }  

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
IF cl$_$get_flag ('-sacl[]', num_args) THEN
BEGIN
    fu_opts := fu_opts - [fu$_$dacl]; { Remove default option }
    fu_opts := fu_opts + [fu$_$sacl]; { Target gets same ACL as source }
END;
IF cl$_$get_flag ('-pdt[]', num_args) THEN { Preserve source's }
    fu_opts := fu_opts + [fu$_$preserve_dt]; { modification and used times}
cl$_$check_unclaimed:
{ Get first argument. }
select := cl$_$first;
WHILE cl$_$get_name ( select, pathname, pathname_len, sizeof(pathname) ) DO BEGIN
    { Get second argument and copy the file. }
    select := cl$_$next;
    IF cl$_$get_derived_name ( new_pathname, new_pathname_len, sizeof(new_pathname) ) THEN BEGIN
        fu$_$copy_file ( pathname, { Source file }
            pathname_len, { Length of source file }
            new_pathname, { Target file }
            new_pathname_len, { Length of target file }
            fu_opts, { FU options in effect }
            fu_context, { Error context }
            status ); { Completion status }
        check_status;
        IF status.all = status$_$ok THEN
            vfmt$_$write2 ('Copied to file named:"%a". ',
                new_pathname, new_pathname_len);
    END; { then begin }
END; { while }
END; { copy_file }
{ Internal Procedure -- Copy_tree -------------------------------------------------- }  
{ Copies source pathname to target pathname. }
PROCEDURE copy_tree:

VAR
    new_tree : name$_$pname_t;
    new_tree_len : integer;
    bef_time : time$_$clock_t;
    aft_time : time$_$clock_t;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
BEGIN

\{ Set verb for query in case user uses wildcards. \}
\setverb ('Copy'. 4);  \{ Verb, length of verb \}

\{ Set wildcard options. \}
\setwildoptions ([\$wild_files, \$wild_dirs, \$wild_exclusive]);

\{ Reinitialize default FU options. \}
\fuopts := ([\$coe, \{ Continue on error \}
   \$print_errors, \{ Print FU errors \}
   \$dacl, \{ Target gets ACL of destination directory \}
   \$subs ]); \{ Retain source ACL for objects in subsystems \}

\{ Get the filenames. \}
\writeln (' Enter the name of the directory you want to copy, then ');
\writeln (' the name of the new directory you want created. ');
\writeln;
\writeln (' Type "-ms[]" if you want to merge the source and target ');
\writeln (' if both are directories. ');
\writeln;
\writeln (' Type "-r[replace]" if you want to replace the target with ');
\writeln (' the source. If target exists, it will be deleted. ');
\writeln (' If it doesn't exist, it will be created. ');
\writeln;
\writeln (' Type "-bef date time" if you want to copy only those files ');
\writeln (' modified before a certain date and time. ');
\writeln;
\writeln (' Type "-aft date time" if you want to copy only those files ');
\writeln (' modified after a certain date. ');
\writeln;
\writeln (' Use this format for date time: yr/month/day hr:min:sec. ');
\writeln (' Example: -bef 85/11/27 12:35:09 source target ');
\writeln;

\ok := \$parse_input ( stdin, null_line );

IF null_line THEN
\writeln ('No input. Type CTRL/Q to quit.');

\{ Get keywords, set appropriate FU options, then check for any invalid keywords; exit if user specified any. \}

\{ Get flags and set appropriate FU options. \}
\furopts := \fuopts + [\$merge]; if they are directories \}
\furopts := \fuopts + [\$replace]; \{ Replace target with source \}
BEGIN
get_time ( bef_time ); \{ Operate on files modified before date \}
END;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
IF cl_$get_flag ('-aft[]', num_args) THEN
BEGIN
    get_time ( aft_time );  \{ Operate on files modified \}
    fu_opts := fu_opts + [fu_$af_time];  \{ after date \}
END;
cl_$check_unclaimed;
{ Get first argument. }
select := cl_$first;
WHILE cl_$get_name ( select, pathname, pathname_len, sizeof(pathname) ) DO BEGIN
{ Get second argument and copy the tree. }
select := cl_$next;
IF cl_$get_derived_name ( new_tree, new_tree_len, sizeof(new_tree) ) THEN BEGIN
    fu_$copy_tree ( pathname, \{ Source pathname \}
        pathname_len, \{ Source pathname length \}
        new_tree, \{ Target pathname \}
        new_tree_len, \{ Target pathname length \}
        fu_opts, \{ Options \}
        bef_time.high, \{ Before time \}
        aft_time.high, \{ After time \}
        fu_error, \{ Error pathname \}
        fu_error_len, \{ Error pathname length \}
        status ); \{ Completion status \}
    check_status;
    IF status.all = status $ok THEN
        vfmt_$write2 ( 'Copied to directory named "%a" %'.
            new_tree, new_tree_len );
END; \{ then begin \}
END; \{ while \}
END; \{ copy_tree \}

{ Internal Procedure -- Delete_file }-------------------------------

PROCEDURE delete_file;
{ Delete specified file. }

VAR
    force : boolean;
    del_unlkd : boolean;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
BEGIN

{ Set wildcard options. }
cl_set_wild_options ([cl_wild_files]);

{ Reinitialize default FU options. }
fu_opts := [{ fu$co, { Continue on error }
  fu_sprint_errors, { Print FU errors }
  fu$dac1, { Target gets ACL of destination directory }
  fu$subs }]; { Retain source ACL for objects in subsytems }

{ Get the filename. }

writeln (' Enter the name of the file(s) you want to delete. ');
writeln (' Type -f[orce] if you want to force the delete. ');
writeln (' Type -du if you want to delete a locked object. ');
ok := cl_parse_input ( stream$stdin, null_line);

IF null_line THEN
  writeln ('No input. Terminating program. ');

{ Get keywords and check for any invalid keywords; exit if user specified any. }

force := cl_get_flag ('-f[orce]', num_args);
del_unlkd := cl_get_flag ('-d[u][]', num_args);

cl_check_unclaimed;

{ Set verb for query in case user uses wildcards. }

cl_set_verb ('Delete', 6); { Verb, length of verb }

{ Get first argument. }

select := cl_first;

WHILE cl_get_name ( select, pathname, pathname_len, sizeof(pathname) )
  DO BEGIN
    fu_delete_file ( pathname, { File to be deleted }
      pathname_len, { Length of file }
      force, { If TRUE, deletes file if user has owner rights, }
      { but not delete rights. }
      del_unlkd, { If TRUE, deletes file even if it is locked. }
      status);

    check_status;
    IF status.all = status$ok THEN
      vfmt$write2 ( 'Deleted file named, "%a", %r.' ,
        pathname, pathname_len);
  END; { while }
END; { delete_file }

Example 2-2. Operating on Files and Trees with FU Calls, Cont.

Handling Files and Trees with FU  2-12  Apollo Confidential
PROCEDURE delete_tree;
{ Delete specified directory. }
BEGIN

{ Set verb for query in case user uses wildcards. }
c1$_set_verb ('Delete', 6); { Verb, length of verb }

{ Set wildcard options. }
c1$_set_wild_options ([c1$_wild_files, c1$_wild_dirs, c1$_wild_exclusive]);

{ Reinitialize default FU options. }
fu_opts := [
  fu$_coe, { Continue on error }
  fu$_print_errors, { Print FU errors }
  fu$_dloc, { Target gets ACL of destination directory }
  fu$_subs ]; { Retain source ACL for Objects in subsystems }

{ Get the filenames. }
writeln;
writeln (' Enter the name of the directory you want to delete. ');
writeln (' Note that you will delete all the files, links and ');
writeln (' subdirectories located in this directory. ');
writeln;
writeln (' Type "-l[ist]" if you want to list all the Objects as ');
writeln (' they are deleted. ');
writeln;
writeln (' Type "-lf" if you want to list only files as they are deleted. ');
writeln;

ok := c1$_parse_input ( stream$_stdin, null_line );

{ Get keywords, set appropriate FU options, then check for any invalid keywords; exit if user specified any. }

IF null_line THEN
  writeln ('No input. Type CTRL/Q to quit.');
IF c1$_get_flag ('-l[ist]', num_args) THEN
  fu_opts := fu_opts + [fu$_list_links, fu$_list_dirs, fu$_list_files];
IF c1$_get_flag ('-lf', num_args) THEN
  fu_opts := fu_opts + [fu$_list_files];

cl$_check_unclaimed;

{ Get first argument. }
select := c1$_first;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
WHILE cl$_get_name ( select, pathname, pathname_len, sizeof(pathname) )
DO BEGIN
  fu$_delete_tree ( pathname, { Source pathname }  
  pathname_len, { Source pathname length }  
  fu_opts, { Options }  
  fu_error, { Error pathname }  
  fu_error_len, { Error pathname length }  
  status ); { Completion status }

  check_status;

  IF status.all = status $ok THEN
    vfmt$_write2 ( 'Delete directory named,"%a".  
    pathname, pathname_len );
  END; { while }
END; { delete_tree }

{ Internal Procedure -- Compare_tree =================================== }  

{ Compare source tree to target tree. }

PROCEDURE compare_tree;

VAR
  tar_pathname : name$_pname_t;  
  tar_pathname_len : integer;
BEGIN

  { Set verb for query in case user uses wildcards. }
  cl$_set_verb ( 'Compare', { Verb }  
  7 ); { Length of verb }

  { Set the wildcard options. }

  cl$_set_wild_options ( [cl$_wild_files, cl$_wild_dirs, cl$_wild_exclusive] );

  { Reinitialize default FU options. }
  fu_opts := ([ fu$_coe, { Continue on error }  
    fu$_print_errors, { Print FU errors }  
    fu$_dacl,       { Target gets ACL of destination directory }  
    fu$_subs ]); { Retain source ACL for objects in subsytems }

  { Get the filenames. }

  writeln ( ' Enter the name of the two directories you want to compare. ' );
  writeln ( ' Type "-l" or "-list" if you want to compare all the directories ' );
  writeln ( ' and files. ' );
  writeln;
  writeln ( ' Type "-if" if you want to compare all the files. ' );
  writeln;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
ok := cl_$parse_input ( stream$stdin, null_line );

IF null_line THEN
    writeln ('No input. Type CTRL/Q to quit.');

{ Get keywords, set appropriate FU options, then check for any invalid
     keywords; exit if user specified any. }

IF cl_$get_flag ('-1[ist]', num_args) THEN
    fu_opts := fu_opts + [fu$_list_links,
                           fu$_list_dirs,
                           fu$_list_files];

IF cl_$get_flag ('-lf[]', num_args) THEN
    fu_opts := fu_opts + [fu$_list_files];

cl$_check_unclaimed;

{ Get first argument. }

select := cl$_first;

WHILE cl$_get_name ( select, pathname, pathname_len, sizeof(pathname) )
     DO BEGIN

{ Get second argument and copy the tree. }

select := cl$_next;

IF NOT cl$_get_derived_name ( tar_pathname, tar_pathname_len,
                               sizeof(tar_pathname) )
     THEN BEGIN

    vfmt$_write2 ( 'No pathname to compare with "%a" %
                     .
                     pathname,
                     pathname_len );
    pgm$_set_severity ( pgm$_error );
    pgm$_exit
    END; { then begin }

    fu$_cmp_tree ( pathname, { Source pathname }
                  pathname_len, { Source pathname length }
                  tar_pathname, { Target pathname }
                  tar_pathname_len, { Target pathname length }
                  fu_opts, { Options }
                  fu_error, { Error pathname }
                  fu_error_len, { Error pathname length }
                  status ); { Completion status }

    check_status;

END; { while }

END; { compare_tree }

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
{ Internal Procedure -- Move_file ------------------------------- }

PROCEDURE move_file; { Move file to another location. } 

VAR 
  tar_pathname : name$pathname_t; 
  tar_pathname_len : integer; 

BEGIN 

{ Set verb for query in case user uses wildcards. } 

  cl$_set_verb ('Move', 4): { Verb, length of verb } 
  cl$_setDerived_count (1): 

{ Set wildcard options. } 

  cl$_setWild_options ([cl$_wild_files]); 

{ Reinitialize default FU option: } 

  fu_opts := ([ fu$_coe, { Continue on error } 
               fu$_print_errors, { Print FU errors } 
               fu$_acl, { Target gets ACL of destination directory } 
               fu$_subs ]); { Retain source ACL for objects in subsytems } 

{ Get the filenames. } 

  writeln (' Enter the name of the file you want to move, then ');
  writeln (' the pathname where you want to move it to. ');
  writeln; 
  writeln (' Type "-l[ist]" if you want to list the files. ');
  writeln (' Type "-r[eplace]" to replace the target file with the ');
  writeln (' source file if it exists. ');
  writeln (' Type "-chn" to change the name of the target file. ');
  writeln (' Type "-fdl" to delete the target file if it exists. ');
  writeln (' if you have owner rights. ');
  writeln;

  ok := cl$_parse_input ( stream$stdin, null_line ); 

  IF null_line THEN 
    writeln ('No input. Type CTRL/Q to quit.'); 

{ Get keywords, set appropriate FU options, then check for any invalid keywords; exit if user specified any. } 

  IF cl$_get_flag ('-l[ist]', num_args) THEN 
    fu_opts := fu_opts + [fu$_list_files]; 
  IF cl$_get_flag ('-r[eplace]', num_args) THEN 
    fu_opts := fu_opts + [fu$_replace]; 
  IF cl$_get_flag ('-chn[]', num_args) THEN 
    fu_opts := fu_opts + [fu$_rename]; 
  IF cl$_get_flag ('-fdl[]', num_args) THEN 
    fu_opts := fu_opts + [fu$_force_del];
Example 2-2. Operating on Files and Trees with FU Calls, Cont.
ok := cl$_sparse_input ( stream$_stdin, null_line );

IF null_line THEN
  writeln ('No input. Type CTRL/Q to quit.');

{ Get first argument. }
select := cl$_first;

WHILE cl$_get_name ( select, pathname, pathname_len, sizeof(pathname) )
  DO BEGIN
    fu$_rename_unique ( pathname, { Source pathname } pathname_len, { Source pathname length } fu_opts, { Options } status ); { Completion status }
  check_status;
  END; { while }
END; { rename_file }

BEGIN { Main }

{ Initialize the CL. but don't parse the command line. }
cl$_setup ( [cl$_no_match_error, { Terminate program when wildcards do not match any existing pathname. } cl$_verify_all], { Verify all names with user. } program_name, { Name of program } sizeof(program_name)); { Length of program name }

{ Initialize FU, set the program name so that FU calls can identify the program name when reporting errors. }
fu$_init;

fu$_set_prog_name ( program_name, sizeof(program_name) );

{ Set up default FU options. }
fu_opts := ([ fu$_coe, { Continue on error } fu$_print_errors, { Print FU errors } fu$_dacl, { Target gets ACL of destination directory } fu$_subs ]); { Retain source ACL for objects in subsytems }

{ Get the line to parse from standard input. }
writeln;
writeln ( 'This program allows you to handle files and trees. ' );
writeln;

Example 2-2. Operating on Files and Trees with FU Calls, Cont.
Example 2-2. Operating on Files and Trees with FU Calls, Cont.
Chapter 3
Logging In and Changing the Registry

The LOGIN Manager is a set of DOMAIN system routines that allows you to tailor a log-in operation or change a user's registry ACCOUNT file. You might want to change a log-in operation to provide stricter log-in requirements. You can change a registry ACCOUNT file to change a user's password or home directory.

Before using the LOGIN system calls, you should be familiar with registry PERSON, PROJECT, ORGANIZATION (PPO) and ACCOUNT files. For more information, see the Administering Your DOMAIN System manual.

3.1. System Calls, Insert Files, and Data Types

To use the LOGIN manager, use the system calls with the prefix LOGIN. This chapter describes how most of these calls work. For details on LOGIN call syntax, data types, and error messages, see Part II of this manual.

When using LOGIN system calls in your program, you must specify the appropriate insert file for the language you are using. The LOGIN insert files are

/SYS/INS/LOGIN.INS.C for C.
/SYS/INS/LOGIN.INS.FTN for FORTRAN.
/SYS/INS/LOGIN.INS.PAS for Pascal.

In addition, you might need to bind your program with the source file, NLOGIN.BIN because the entry points to most LOGIN system calls are not in a global library.

3.2. Overview of the LOGIN System Calls

The LOGIN subsystem contains system calls that allow you to write a tailored log-in procedure, or change a user's password or home directory.

To tailor a log-in operation, use the system call, LOGIN_$LOGIN.

To change a user's registry ACCOUNT file, use the LOGIN system calls:

- LOGIN_$OPEN
- LOGIN_$CHPASS
- LOGIN_$CHHDR
- LOGIN_$CKPASS
- LOGIN_$CLOSE

Sections 3.3 and 3.4 describes the two log-in operations.
3.3. Tailoring a Log-In Operation

Use the LOGIN_ $LOGIN system call to tailor a LOGIN operation. For example, you can require users to have different PPO identities to access a particular program. That is, the user DARLENE.MARKETING might need to log in as DARLENE.MARKETING.NEW_PRODUCTS.SPHINX to have full rights to an object.

To write your own LOGIN procedure using the LOGIN_ $LOGIN system call, you supply LOGIN_ $LOGIN with an open stream, and pointers to your own log-in I/O routines. Your program passes an open stream to LOGIN_ $LOGIN, LOGIN_ $LOGIN checks the password the user supplied against the registry's password for that PPO. If they match, the program logs the user in.

NOTE: Your program must be a protected subsystem in the LOGIN subsystem. (See the DOMAIN System User's Guide for more information on protected subsystems.)

Since LOGIN_ $LOGIN is designed to be device-independent, you must supply your own input/output routines. LOGIN_ $LOGIN requires four routines: a read, write, help, and an open log routine. LOGIN_ $LOGIN uses these routines to get the user's input and display messages.

Example 3-1 shows how you declare pointers to your routines. The second parameter, LOGIN_ $LOG_EVENTS, indicates that you are supplying an open log routine so that LOGIN_ $LOGIN records any user's attempts to log in.

To get the addresses of these routines in Pascal, you must put the I/O routines in a module separate from your main log-in program. Bind the two binary files to execute your log-in program.

```pascal
{ Get addresses of external I/O procedures for LOGIN_ $LOGIN. }

login_procedures.pread := addr(my_read);
login_procedures.pwrite := addr(my_write);
login_procedures.help := addr(my_help);
login_procedures.open_log := addr(my_open_log);

IF NOT login_ $login ( stream_in, { Stream ID where user will log in } [login_ $log_events], { Logging events } login_procedures, { Address of your I/O routines } status ) { Completion status }
THEN
  IF (status.all = login_ $err_shut) THEN
  BEGIN
    err_exit(status);
  END;

Example 3-1. Declaring External I/O Routines for LOGIN_ $LOGIN

You are not restricted in how you write your I/O routines, except that each routine's parameters must correspond to those arguments that LOGIN_ $LOGIN expects. The routines must also perform the operation that LOGIN_ $LOGIN expects:
• MY_READ gets the PPO string and password for logging in. It also prompts for a new password if the user types ‘-P’, or for a new home directory if the user types ‘-H’.* If MY_READ does not provide a way to get this information, LOGIN_$LOGIN prompts for it.

• MY_WRITE writes any error messages to output.

• MY_HELP provides a message for when a user types ‘-h[elp]’ at the log-in prompt.

• MY_OPEN_LOG records all successful and unsuccessful log-in attempts if you specified the LOGIN_$LOG_EVENTS option when you called LOGIN_$LOGIN.

When you specify LOGIN_$LOG_EVENTS, LOGIN_$LOGIN calls your log event routine at each log-in attempt. It passes the name of its log file to your routine, opens the file, records the event, and closes the file. LOGIN_$LOGIN will record both successful and unsuccessful log-in attempts in the file, ‘NODE_DATA/SIOLOGIN_LOG’. Your log-in routine can create its own log file simply by ignoring the file name that LOGIN_$LOGIN passed. However, note that LOGIN_$LOGIN will close your file after each log-in attempt.

Example 3-2 shows how to write the parameters for the I/O routines in Pascal.

```
{ Login read procedure ---------------------------------------- }

FUNCTION my_read (IN stream : stream_id_t; { Stream ID on which to log user in, usually stream_stdin })
    OUT inbuf : UNIV login_string_t; { Returns string to supply to LOGIN_$LOGIN }
    IN inlen : integer; { Maximum length of string buffer }
    IN pstr : UNIV login_string_t; { Prompt string }
    IN plen : integer; { Length of ‘pstr’ }
    IN echo : boolean; { Indicates whether to echo characters }
                   { Used to prevent the OM from displaying a password. }
    IN fillbuf : UNIV login_string_t; { Not used, specific to the OM }
    IN fillbuflen : integer { Not used, specific to the OM }
    ). : integer;
EXTERN:

{ Login write procedure --------------------------------------- }

PROCEDURE my_write (IN stream : stream_id_t; { Stream ID on which to display user’s output, usually stream_stdout })
    IN pstr : UNIV login_string_t; { Message to write }
    plen : integer { Length of message }
);
EXTERN:
```

Example 3-2. Writing LOGIN External I/O Routines for LOGIN_$LOGIN
Example 3-2. Writing External I/O Routines for LOGIN_$LOGIN, Cont.

3.3.1. Sample Program Using LOGIN_$LOGIN

Examples 3-3 and 3-4 show how to use the LOGIN_$LOGIN system call in Pascal. The first example is a main procedure that performs a log-in operation. The second example is a module containing the input/output routines for LOGIN_$LOGIN.

The main program gets the addresses of the external I/O routines, and calls LOGIN_$LOGIN. Note that when calling LOGIN_$LOGIN, the LOGIN_$LOG_EVENTS option was specified, so that LOGIN_$LOGIN will record each log-in attempt.

LOGIN_$LOGIN calls MY_READ when it needs log-in information – the login string and password, and, if the user typed a "-P," or "-H," the new password or home directory respectively. It calls MY_WRITE when it needs to write error messages. It calls MY_HELP when the user asked for help at the prompt. It calls MY_OPEN_LOG, (because we set the LOGIN_$LOG_EVENTS option), after each log-in attempt. When calling MY_OPEN_LOG, it records whether the attempt was successful or not, then closes the file.

Each call to MY_OPEN_LOG passes an open stream to LOGIN_$LOGIN.

The external procedures, located in MODULE LOGIN_PROCEDURES (Example 3-4), must be bound with LOGIN_LOGIN (Example 3-3) before the log-in program can be executed.
```pascal
PROGRAM login_login;

{%noiis;
{%include '/sys/ins/base.ins.pas';
{%include '/sys/ins/pfm.ins.pas';
{%include '/sys/ins/streams.ins.pas';
{%include '/sys/ins/vfmt.ins.pas';
{%include '/sys/ins/error.ins.pas';
{%include '/sys/ins/pgm.ins.pas';
{%include '/sys/ins/cl.ins.pas';
{%include '/sys/ins/login.1ns.pas';
{%list;

CONST
  prog_name = 'login_login';

VAR
  cl_opts : cl_sopt_set_t;
  inv_prog : string;
  inv_prog_len : integer;
  cleaner : pfm_$cleanup_rec;
  status : status_$t;
  login_procedures : login_$proc_rec_t;
  stream_in : stream_$id_t := stream_$stdin;
  conv : stream $conv := [stream_$stdin,
      stream_$stdout,
      stream_$errin,
      stream_$errout];
  handle : pgm_$proc;

{ Login read procedure ----------------------------- }
{ LOGIN_$_LOGIN gets input via this procedure. }

FUNCTION my_read(
  IN stream : stream_$id_t; { Stream ID where user logs in }
  OUT inbuf : UNIV login_$string_t; { Login string to pass to LOGIN }
  IN inlen : integer; { Maximum length of string buffer }
  IN pstr : UNIV login_$string_t; { Prompt string }
  IN plen : integer; { Length of prompt string }
  IN echo_mode : boolean; { Indicates whether to echo characters on input pad, used to prevent password echo }
  IN fillbuf : UNIV login_$string_t; { Not used, specific to DM }
  IN fillbuflen : integer { Not used, specific to DM }
  ) : integer;
EXTERNAL;

{ Login write procedure ----------------------------- }
{ LOGIN_$_LOGIN writes output via this procedure. }

PROCEDURE my_write(
  IN stream : stream_$id_t; { Stream ID for user's output }
  IN pstr : UNIV login_$string_t; { Message to write to output }
  IN plen : integer { Length of message }
  );
EXTERNAL;
```

Example 3-3. Performing a Log-In Operation with LOGIN_$_LOGIN

Apollo Confidential 3-5 Logging In and Changing the Registry
{ Login help procedure }

{ LOGIN_$LOGIN calls this procedure when a user unsuccessfully attempts to log in. MY_HELP supplies the message by passing the stream ID of the help file to LOGIN_$LOGIN. }

PROCEDURE my_help(
    IN stream : stream_id_t { Stream ID on which to write help message }
);
EXTERN;

{ Login open log procedure }

{ LOGIN_$LOGIN calls this procedure whenever a user attempts to log in, if the LOGIN_$LOG_EVENTS option was set. }

FUNCTION my_open_log(
    IN log_file : UNIV login_string_t; { Pathname of log file }
    IN logrlen : integer; { Length of "log_file" }
    OUT logstr : stream_id_t { Returns stream ID of log file }
) : boolean; -- { Returns TRUE if successful }
EXTERN;

{ Exit on error INTERNAL procedure }

PROCEDURE err_exit (IN status : UNIV status_t):

VAR
    stat : status_t;
BEGIN
    IF status.all <> status.$ok THEN
        BEGIN
            error.$std_format(status, '%$');
            pgm.$set_severity(pgm.$error);
            pfm.$rls_cleanup(cleaner, stat); {Reset cleanup handler}
            pgm.$exit;
        END;
    END; { err_exit }
BEGIN { Main }

{ Set CL options and initialize the Command Line Handler. }

cl_opts := [cl.$no_wildcards, cl.$no_keyword_delim];
cl.$init (cl_opts, prog_name, sizeof(prog_name));
status := pfm.$cleanup(cleaner);

IF status.all <> pfm.$cleanup_set THEN
    BEGIN
        pgm.$set_severity(pgm.$error);
        pgm.$exit;
    END;

Example 3-3. Performing a Log-In Operation with LOGIN_$LOGIN, Cont.
Example 8-3. Performing a Log-In Operation with LOGIN_$LOGIN, Cont.
MODULE login_procedures;

{ This contains I/O procedures for login.pas. }

%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/cl.ins.pas';
%include '/sys/ins/pad.ins.pas';
%include '/sys/ins/cal.ins.pas';
%include '/sys/ins/time.ins.pas';
%include '/sys/ins/login.ins.pas';
%list;

CONST
    newline = chr(10);
    back_sp = chr(8);
    car_ret = chr(13);
    log_name = 'login_log_file';

VAR
    stream_in : stream_id_t := stream$stdin;
    stream_out : stream_id_t := stream$stdout;
    status : status$t;

{ Login read function --------------------------------------------- }

{ This procedure supplies LOGIN_$LOGIN with user input. }

FUNCTION my_read(
    IN stream : stream_id_t;
    OUT inbuf : UNIV login_$string_t;
    IN inlen : integer;
    IN pstr : UNIV login_$string_t;
    IN plen : integer;
    IN echo : boolean;
    IN fillbuf : UNIV login_$string_t;
    IN fillbuflen : integer
): integer;

VAR
    my_buffer : array[1..4] of char;
    inptr : 'string';
    size : integer32;
    seek_key : stream$sk_t;
    i : integer;
    status : status$t;

BEGIN { my_read }

{ Prompt }
stream$put_chr (stream_out, addr(pstr), plen, seek_key, status);

Example 3-4. Providing I/O Routines for LOGIN_$LOGIN
{ Turn off echo on user's input pad. }

IF NOT echo THEN
BEGIN

    pad_$raw (stream_in, status );  { Echo off }
    i := 1;

    REPEAT { Get input from user }
        stream$_get_rec(stream_in, addr(my_buffer), 1, inptr, size, seek_key, status );
        IF status.all <> status_ok THEN exit;
        IF inptr^1[i] = back_sp THEN
            IF i > 1 THEN i := i - 1;
            next;
        END;
        inbuf[i] := inptr^1[i];
        IF (inbuf[i] = car_ret) OR (inbuf[i] = newline) THEN exit;
        i := i + 1;
    UNTIL FALSE;

    inbuf[i] := newline;
    my_read := i;  { Count includes NEWLINE }

    { Allow pad to echo input again. }
    pad_$cooked(stream_in, status);

    { Write a NEWLINE after message. }
    stream$_put_rec(stream_out, addr(newline), 1, seek_key, status );
END { NOT echo_mode }
ELSE BEGIN { echo_mode }
    { Get input from user }
    stream$_get_rec(stream_in, addr(inbuf), inlen, inptr, size, seek_key, status );
    IF status.all <> O THEN
        my_read := 0
    ELSE my_read := size;
    END; { echo_mode }

    return;
END; { my_read }

Example 3-4. Providing I/O Routines for LOGIN$_LOGIN, Cont.
{ Login write procedure ---------------------------------------- }
{ Writes output }

PROCEDURE my_write(
    IN stream : stream$_id_t;
    IN pstr   : UNIV login$_string_t;
    IN plen   : integer);

VAR
    status    : status_t;
    seek_key  : stream$sk_t;

BEGIN
    stream$sput_chr(stream_out, addr(pstr), plen, seek_key, status);
    stream$sput_rec(stream_out, addr(newline), 1, seek_key, status);
    return;
END; { my_write }

{ Login help procedure ---------------------------------------- }
{ This is the message that gets typed when user types h[elp] at the
  Login prompt. }

PROCEDURE my_help(
    IN stream : stream$_id_t);

CONST
    helpline1 = ' This is the help message that';
    helpline2 = ' gets displayed when the user';
    helpline3 = ' types h[elp] at the prompt. ';
    helplen1 = 31;
    helplen2 = 30;
    helplen3 = 28;

VAR
    status    : status_t;
    seek_key  : stream$sk_t;

BEGIN
    my_write( stream_out, helpline1, helplen1);
    my_write( stream_out, helpline2, helplen2);
    my_write( stream_out, helpline3, helplen3);
END; { my_help }

{ Login open log procedure ---------------------------------------- }
{ Open the log file, write the date and time in it, and return the stream ID. }

FUNCTION my_open_log(
    IN log_file : UNIV login$_string_t;
    IN log_flen : integer;
    OUT logstr  : stream$_id_t)
    : boolean;

CONST
    my_log_file = 'LOGIN_log_file';
    my_log_flen = 14;}
VAR
td : cal$_timedate_rec_t 

da : ARRAY [1..10] OF char 

mon : ARRAY [1..10] OF char 
nd : cal$_weekday_t

st : status$_t;

BEGIN { My_open_log } my_open_log := FALSE;

{ Ignore the log file passed by LOGIN$_LOGIN, and open my own logging file. Return its stream ID to LOGIN$_LOGIN. (If I don't specify log_file LOGIN$_LOGIN writes to node_data/siologin_log.) }

stream$_create(my_log_file, my_log_flen, stream$_append, stream$_no_conc_write, logstr, st);

IF st.all <> 0 THEN return;

{ Get decoded local time. }
cal$_decode_local_time(td); { Time stamp }

WITH td DO BEGIN

d := cal$_weekday (year, month, day);
CASE d OF

  cal$_sun : da := 'Sunday';
  cal$_mon : da := 'Monday';
  cal$_tue : da := 'Tuesday';
  cal$_wed : da := 'Wednesday';
  cal$_thu : da := 'Thursday';
  cal$_fri : da := 'Friday';
  cal$_sat : da := 'Saturday';
  OTHERWISE da := 'Doomsday';
END; { case day }

CASE month OF

  1 : mon := 'January';
  2 : mon := 'February';
  3 : mon := 'March';
  4 : mon := 'April';
  5 : mon := 'May';
  6 : mon := 'June';
  7 : mon := 'July';
  8 : mon := 'August';
  9 : mon := 'September';
  10: mon := 'October';
  11: mon := 'November';
  12: mon := 'December';
END; { case month }

vfmt$_sw10(logstr, '%a, %a %wd, %wd %2wd: %2wd% '.
da, 10, mon, 10, day, year, hour, minute, second, 0);

END; { WITH td }

my_open_log := TRUE;

END; { my_open_log }

Example 3-4. Providing I/O Routines for LOGIN$_LOGIN, Cont.
3.4. Changing User Account Files in the Network Registry

The LOGIN manager provides a set of LOGIN system calls that allow you to edit user account files in the network registry. Network account files associate usernames with log-in passwords and home directories. For details on the network registry, see the Administering Your DOMAIN System manual.

When using LOGIN system calls, you must follow a specific order. Figure 3-1 shows the sequence of using these calls.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOGIN System Call</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize LOGIN.</td>
<td>LOGIN_$OPEN</td>
<td>Sets up storage for LOGIN internal datatypes, and assigns a LOGIN pointer to that storage.</td>
</tr>
<tr>
<td>Set the PPO to user whose account file you will change.</td>
<td>LOGIN_$SET_PPO</td>
<td>Converts the PPO string to internal form, and &quot;remembers&quot; it.</td>
</tr>
<tr>
<td>Check user password (optional).</td>
<td>LOGIN_$CKPASS</td>
<td>Checks the registry to see if the given password is the same as the one listed for the current PPO.</td>
</tr>
<tr>
<td>Change user password.</td>
<td>LOGIN_$CHPASS</td>
<td>Changes the password of the current PPO to the specified password.*</td>
</tr>
<tr>
<td>Change user home directory.</td>
<td>LOGIN_$CHHDIR</td>
<td>Changes the home directory of the current PPO to the specified home directory.*</td>
</tr>
<tr>
<td>Do you want to change anymore users?</td>
<td></td>
<td>* Note that changes are not finalized until LOGIN_$CLOSE closes successfully.</td>
</tr>
<tr>
<td>Yes</td>
<td>LOGIN_$CLOSE</td>
<td>Updates registry ACCOUNT files, and releases storage.</td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Note that changes are not finalized until LOGIN_$CLOSE closes successfully.

Figure 3-1. Using LOGIN System Calls in Proper Sequence
When you initialize the LOGIN manager with LOGIN_$OPEN, it allocates heap storage for its private datatypes and returns a pointer value to that storage. Use the LOGIN pointer as an input parameter in subsequent LOGIN system calls.

When you open LOGIN, specify either of two modes: LOGIN_$READ or LOGIN_$UPDATE. LOGIN_$READ allows you to read the user's password but you cannot change it. LOGIN_$UPDATE allows you to change a user's account file. Note that to change a user's account file, you must be a protected subsystem.

After initializing the LOGIN manager, set the PPO to the user whose account file you want to view or change with LOGIN_$SET_PPO. A user's PPO has four fields: the user's name, project name, organization name, and the hexadecimal ID of the user's node. The following are two examples of PPO's: WALLY_W.CURRENT.WRITERS.2713, ZACH.NONE.NONE.3541.

If you do not set the PPO with this call, the LOGIN manager will automatically set the PPO to the user who is currently logged in. Even if you want the current user, it's good programming practice to set the PPO explicitly with this call. You can specify the current user efficiently by specifying a null PPO and a PPO length of zero. LOGIN_$SET_PPO sets the PPO to the current user.

To check a user's password use LOGIN_$CKPASS, specify the LOGIN pointer (returned by LOGIN_$OPEN), the password you want to check, and its length. It returns the error message LOGIN_$BAD_PASSWD if the password check failed.

To change a user's password use LOGIN_$CHPASS, specifying the LOGIN pointer (returned by LOGIN_$OPEN), the new password, and its length.

To change a user's home directory, use LOGIN_$CHDIR, specifying the LOGIN pointer (returned by LOGIN_$OPEN), the new home directory, and its length.

Use LOGIN_$CLOSE to close the LOGIN manager. At this time, the LOGIN manager updates the account files in the network registry. If a previous LOGIN system call had difficulty with an account file, a LOGIN_$CLOSE can fail, which means the registry will not be updated accurately. You will have to repeat the entire LOGIN sequence since you opened LOGIN with LOGIN_$LOGIN. Use LOGIN_$CLOSE even if LOGIN_$OPEN failed, because when LOGIN_$OPEN fails it can leave LOGIN partially opened.

3.4.1. Sample Program -- Changing the Registry ACCOUNT Files

Example 3-5 shows how to use the various LOGIN system calls to check and change a password and to change a home directory.
PROGRAM login_change_pass_dir;

%noist;
#include '/sys/ins/base.ins.pas';
#include '/sys/ins/vfmt.ins.pas';
#include '/sys/ins/error.ins.pas';
#include '/sys/ins/pgm.ins.pas';
#include '/sys/ins/cl.ins.pas';
#include '/sys/ins/login.ins.pas';
%list;

CONST
  prog_name = 'LOGIN_chng_pass_dir';

VAR
  command : string;
  command_len : integer;
  ok : boolean;
  null_line : boolean;
  ppo : string;
  ppo_len : integer;
  verify : boolean;
  num_args : integer;
  login_ptr : login_$ptr;
  bad_pname : name_$pname_t;
  bad_pname_len : integer;
  errout : stream_$id_t := stream_$errout;
  err_status : status_$t;
  status : status_$t;

{ ***************************************************************** }
PROCEDURE check_err ( IN status : status_$t;
                      IN msg : UNIV string );
BEGIN
  IF (status.code = 0 ) THEN
    RETURN;
    error_$std_format (status, msg);
    pgm_$set_severity (pgm_$error);
    pgm_$exit
  END; { check_err }

{ ***************************************************************** }
PROCEDURE chn_pass: { Go here if user specified chpass }
{ If the user specifies the -v[erify] option with a password, this
  procedure checks the specified password; otherwise, it changes the
  user's password to the specified password. }
VAR
  old_pass : string;
  new_pass : string;
  opass_len : integer;
  npass_len : integer;

Example 3-5. Using LOGIN Calls to Change ACCOUNT Files
BEGIN { chn_pass }

{ Get password. If user wants to verify current password, the next argument is the password to check. If the user wants to change the password, the next argument is the new password. }

IF verify THEN
  ok := cl$_get_arg (cl$_next, old_pass, opass_len, sizeof(old_pass))
ELSE
  ok := cl$_get_arg (cl$_next, new_pass, npass_len, sizeof(new_pass));
cl$_check_unclaimed;

{ If verifying old password, open LOGIN for read access, since you are not changing the registry. }

IF verify THEN
BEGIN
  login$_open (login$_read, { Open LOGIN manager for updates }
    login_ptr, { Pointer to LOGIN datatypes }
    status); { Completion status }
  check_err (status, ' Cannot perform LOGIN open. %$');

{ Set the PPO to operate on. }
  login$_set_ppo (login_ptr, { Pointer to LOGIN datatypes }
    ppo, { PPO file specified by user }
    ppo_len, { Length of PPO file }
    status ); { Completion status }
  check_err (status, ' Cannot set user''s PPO. %$');
{ Check old password. }
  login$_ckpass (login_ptr, { Pointer to LOGIN datatypes }
    old_pass, { Old password }
    opass_len, { Length of old password }
    status); { Completion status }

IF status.all = login$_bad_passwd THEN BEGIN
  vfmt$_ws2(errout, 'Old password is incorrect.I'. O. 0);
  pgm$_set_severity (pgm$_false);
  pgm$_exit;
END
ELSE vfmt$_write2(' Password verified.I'. O. 0);
ELSE BEGIN { Change current password to specified password. }
  login$_open (login$_update, login_ptr, status);
  check_err (status, ' Cannot perform LOGIN open. %$');

{ Set the PPO to operate on. }
  login$_set_ppo (login_ptr, { Pointer to LOGIN datatypes }
    ppo, { PPO file specified by user }
    ppo_len, { Length of PPO file }
    status ); { Completion status }
  check_err (status, ' Cannot set user''s PPO. %$');
  login$_chpass (login_ptr, { Pointer to LOGIN datatypes }
    new_pass, { New password }
    npass_len, { Length of new password }
    status ); { Completion status }

Example 3-5. Using LOGIN Calls to Change ACCOUNT Files, Cont.
IF status.all <> status_ok THEN
BEGIN
  login_err_context (login_ptr, { Pointer to LOGIN datatypes })
  err_status, { Returns error status }
  bad_pname, { Pathname of user that failed }
  bad_pname_len, { Length of bad pathname }
  status); { Completion status }

  vfmt_ws2 (errout, 'Cannot change password for: ',
    bad_pname, sizeof(bad_pname));

END;

END;

login_close (login_ptr, status);
check_err (status, 'Cannot perform LOGIN close. %$');
END; { chn_pass }

PROCEDURE chn_dir; {************************************************ }
{ This procedure changes the user's home directory listed in }
{ the user's ACCOUNT file to the specified home directory. }

VAR
  new_dir : string;
  dir_len : integer;

BEGIN { chn_dir }
{ Get the new home directory. }
  ok := cl_get_arg (cl_next, new_dir, dir_len, sizeof(new_dir));
  cl_check_unclaimed;

  login_open (login_update, { Open LOGIN manager for updates }
    login_ptr, { Pointer to LOGIN datatypes }
    status); { Completion status }
  check_err (status, 'Cannot perform LOGIN open. %$');

{ Set the PPO to operate on. }
  login_set_ppo (login_ptr, { Pointer to LOGIN internal data }
    ppo, { PPO }
    ppo_len, { Length of PPO }
    status); { Completion status }
  check_err (status, 'Cannot set user's PPO. %$');

  login_setchdir (login_ptr, { Pointer to LOGIN internal data }
    new_dir, { Name of new home directory }
    dir_len, { Length of new home directory }
    status); { Completion status }

IF status.all <> status_ok THEN
BEGIN
  login_err_context (login_ptr, { Pointer to LOGIN datatypes }
    err_status, { Returns error status }
    bad_pname, { Pathname of user that failed }
    bad_pname_len, { Length of bad pathname }
    status); { Completion status }

Example 3-5. Using LOGIN Calls to Change ACCOUNT Files, Cont.
VFMT $WS2 (errout, ' Cannot home directory for: ',
bad_pname, sizeof(bad_pname));

END;

login_close (login_ptr, status);
check_err (status, ' Cannot perform LOGIN close. %$');

END: { chn_dir }

{ *********************************************** }

BEGIN { Main }

{ Initialize the CL, but don't parse the command line. }

c1_setup ( [ ],

{ Default CL options in effect }

prog_name, { Name of program }

sizeof(prog_name)); { Length of program name }

{ Get the line to parse from standard input. }

writeln;
writeln ( ' **** ' );
writeln;
writeln ( ' This program allows you to change a user's ' );
writeln ( ' password or home directory. ' );
writeln;

REPEAT { until user types CTRL/Q }

writeln;
writeln ( ' Enter the user's PPO whose password or home ' );
writeln ( ' directory you want to change. ' );
writeln;

{ Get the PPO. }
ok := cl_parse_input ( stream$stdin, null_line );
ok := cl_get_arg ( cl_first, ppo, ppo_len, sizeof(ppo));

writeln;
writeln ( ' Type ' chpass -v[erify] 'password ' if you want to check');
writeln ( ' the current password. ' );
writeln;
writeln ( ' Type ' chpass 'password' to change the current password');
writeln ( ' to the specified password. ' );
writeln;
writeln ( ' Type ' chdir 'dir' to change current home directory');
writeln ( ' to the specified home directory. ' );

Example 3-5. Using LOGIN Calls to Change ACCOUNT Files, Cont.
{ Get the operation. }

ok := cl$_$parse_input ( stream$_$stdin, null$_$line );

{ Check if user supplied the verify option on the command line. Check for any invalid keywords; exit if user specified any. }

verify := cl$_$get_flag ('-v[erify]', num_args);

c1$_$check_unclaimed;

{ Get the operations. }

IF cl$_$get_arg (cl$_$first, command, command_len, sizeof(command)) THEN BEGIN

IF cl$_$match ('chpass[]', command, command_len) THEN chn_pass;

IF cl$_$match ('chhd1r[]', command, command_len) THEN chn_dir;

END;

UNTIL FALSE; { User types CTRL/Q }

END. { Main }

Example 3-5. Using LOGIN Calls to Change ACCOUNT Files, Cont.
Chapter 4
More Process Manager System Calls

This chapter describes DOMAIN system calls for managing programs. These calls allow you to

- Load and call a program or library object module.
- Assign a name given its process UID.
- Set the priority given its process UID.

These system calls are part of the DOMAIN subsystems for managing programs. For details on other calls in these subsystems, see the chapter on managing programs in the Programming with General System Calls manual.

4.1. System Calls, Insert Files, and Data Types

To load and call object modules or libraries, use the LOADER manager. These calls are located in the LOADER insert file. Note that they have the PM prefix.

To assign a name to a given process UID, use PM_SET_NAME, which is in the PM insert file.

To set the priority of a given process UID, use PROC_SET_PRIORITY, which is in the PROC2 insert file.

When using these system calls in your program, you must specify the appropriate insert file for the language you are using. Where prefix is the desired subsystem, the insert files are

/SYS/INS/prefix.INS.C for C.
/SYS/INS/prefix.INS.FTN for FORTRAN.
/SYS/INS/prefix.INS.PAS for Pascal.

For details on the system call syntax, data types, and error messages, Part II of this manual. For details on other system calls in these subsystems, see the DOMAIN System Call Reference manual.

4.2. Loading and Calling a Program with LOADER System Calls

The two system calls in the LOADER manager allow you to load a program or library object module into memory so that it can be executed. PM_LOAD converts a specified object module and returns its starting address. PM_CALL invokes the loaded object module.

These calls work similarly to PGM_INVOKE, except that PGM_INVOKE performs some operations automatically. Table 4-1 highlights the differences between PGM_INVOKE and the LOADER calls. The resources referred to in this table include read/write storage, mark release handlers, address space, and stream descriptors.

For details on using PGM_INVOKE, see the Programming with General System Calls manual.
Table 4-1. Difference Between LOADER and PGM_$INVOKE System Calls

<table>
<thead>
<tr>
<th>Condition</th>
<th>A Loads &amp; Calls B (PM)</th>
<th>A Invokes B (PGM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Level:</td>
<td>Not created.</td>
<td>Created.</td>
</tr>
<tr>
<td>Resources that B</td>
<td>Released when</td>
<td>Released when</td>
</tr>
<tr>
<td>acquires:</td>
<td>A terminates.*</td>
<td>B terminates.</td>
</tr>
<tr>
<td>Storage unmapped:</td>
<td>When A terminates.</td>
<td>When B terminates.</td>
</tr>
</tbody>
</table>

* Note that if A continues to load programs that terminate abnormally, you can run out of resources (such as stream descriptors or address space) since they don’t get released until A terminates.

PM_$LOAD and PM_$CALL are most commonly used to load libraries. You would also use them when you want to control how the object module gets loaded. For example, you can have PM_$LOAD make a writable copy of the object module so that you can set breakpoints without changing the original object module. Or you can have PM_$LOAD report an error if the object module contains any unresolved global variables.

When you use PM_$LOAD, you control how the object module gets loaded by setting the options of a variable in PM_SLOADER_OPTS format. Table 4-2 lists the options you can specify.

Table 4-2. PM_$LOAD Options

<table>
<thead>
<tr>
<th>PM_$LOAD Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_$COPY_PROC</td>
<td>Copies the object module into read/write storage so that you can write to the object module without changing the original object module. If you do not specify this option, PM_$LOAD maps the object module directly.</td>
</tr>
<tr>
<td>PM_$INSTALL</td>
<td>Indicates that you are loading a library. PM_$LOAD makes all the global entry points, which were marked at binding, available to other programs. Specify this option when loading libraries.</td>
</tr>
<tr>
<td>PM_$INSTALL_SECTIONS</td>
<td>Indicates that you are loading a library containing global sections. PM_$LOAD makes these global sections, which were marked at binding, available to all programs. Specify this option when loading libraries.</td>
</tr>
<tr>
<td>PM_$LOAD_WRITABLE</td>
<td>Allows you to write to the object module. Specify this option when you want to write to the object module without making a copy of it first. The debugger provides this with its -NC option.</td>
</tr>
<tr>
<td>PM_$NO_UNRESOLVEDS</td>
<td>Returns an error message if the object module contains any unresolved global variables.</td>
</tr>
</tbody>
</table>
After you load the program with PM_$LOAD, use PM_$CALL to invoke it. When using PM_$CALL, you give it a pointer to the routine you want to call. This is the start address of the object module returned by PM_$LOAD, which is the first field in the predefined record, PM_$LOAD_INFO.

Section 4.2.1 is a sample program using PM_$LOAD and PM_$CALL to load and invoke a program.

Even though PM_$CALL is a function, you usually load a procedure, so the value it returns is meaningless. However, you can use PM_$CALL to return a value from the object module; PM_$CALL will pass a 32-bit integer value along to the main program as its return value. Note that you cannot pass any arguments to the routine you are invoking.

You can have PM_$CALL return data types other than a 32-bit integer, so long as the data fits in 32 bits or fewer. To do so, create a variant record with a 32-bit integer as one variant, and the actual data type returned by the target routine as another variant. Pad the second variant out to 32 bits. Assign the 32-bit integer value from PM_$CALL to the 32-bit integer variant, and take your desired data from the other variant. Example 4-1 shows how you would use PM_$CALL with a variant record.

{ This Pascal example uses PM_$CALL to call a function that returns a 16-bit integer data item: }

VAR
  ret: RECORD CASE INTEGER OF
    1: (long : integer32);
    2: (pad : Integer; wanted_I : Integer);  { Filler }
    { Desired data }
END;
load_info : pm_$load_info;
BEGIN
  { Load my_integer16_function. }

  { Call my_integer16_function using the start address returned by the PM_$LOAD_INFO data type in PM_$LOAD. }
  ret.long := pm_$call(load_info.start_addr);

  { Write out the returned value as a 16-bit integer. }
  writeln (ret.wanted_I);

Example 4-1. Returning a 16-Bit Value with PM_$CALL
4.2.1. Sample Programs Using LOADER System Calls

Example 4-2 is a sample program that loads and calls a second program (PM_LOAD_THIS.BIN) using PM_$LOAD and PM_$CALL. It gets the load options from the command line.

```
PROGRAM pm_load;

%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/cl.ins.pas';
%include '/sys/ins/loader.ins.pas';
%include '/sys/ins/vfmt.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%include '/sys/ins/error.ins.pas';
%list;

CONST
  prog_name = 'pm_load';
  prog_to_load = 'pm_load_this.bin';

VAR
  name : name$pname_t;
  len : integer;
  num_args : integer;
  load_opts : pm$loader_opts;
  sec_info : pm$load_info;
  status : status$t;
  pm_status : status$t;
  ok : boolean;
  null_line : boolean;
  junk : integer32;

{ ---------------------------------------------------------- }

PROCEDURE check_status;

BEGIN
  IF status.all <> status$ok THEN
    BEGIN
      error$print (status);
      pgm$exit;
    END;
  END; { check_status }

{ ---------------------------------------------------------- }
```

Example 4-2. Loading and Calling a Program
BEGIN { Main }

cl_$setup ( [], { Default CL options in effect }
   prog_name, { Name of program }
   sizeof(prog_name)); { Length of program name }

{Initialize empty set of loader options. }
load_opts := [];

writeln ( 'This program loads the object module, ',
   prog_name : sizeof(prog_name) );
writeln;
writeln ( 'You can specify the following options: ' );
writeln;
writeln ( '-c[opy] to copy the object module so you can write to it. ' );
writeln ( '-w[rite] to make the original object module writable. ' );
writeln ( '-nr[esolveds] to terminate program with an error if global variables are unresolved. ' );
writeln ( '-lib[rary] to load a library. ' );
writeln;
ok := cl_$parse_input ( stream_$stdin, null_line );

{ Get keywords, then check for any invalid keywords; exit if user specified any. }

IF cl_$get_flag ('-c[opy]', num_args)
   THEN load_opts := [pm_$copy_proc];
IF cl_$get_flag ('-w[rite]', num_args)
   THEN load_opts := [pm_$load_writable];
IF cl_$get_flag ('-nr[esolveds]', num_args)
   THEN load_opts := [pm_$no_unresolveds];
IF cl_$get_flag ('-lib[rary]', num_args)
   THEN load_opts := [pm_$install + pm_$install_sections];

pm_$load ( prog_to_load, { Name of program to load }
   sizeof(prog_to_load), { Length of program }
   load_opts, { Load options }
   0, { Get start address only }
   sec_info, { Returns start address of object module }
   status); { Completion status }
check_status;

IF status.all = status_$ok THEN
BEGIN
   writeln;
   writeln ('Program loaded sucessfully. ');
   junk := pm_$call( sec_info.start_addr);
END;
END. { pm_load }

Example 4-2. Loading and Calling a Program, Cont.
4.3. Setting a Process Priority

The process priority is an integer ranging from 1 (low) to 16 (high). When the operating system decides which process to run next, it chooses the process that currently has the highest priority.

The priority changes while a process executes. Its priority increases as the process waits for events, and decreases as it computes for long periods without waiting. To find out the priority range of a running process, use the PST (process_status) Shell command.

By default, a process will have a priority range of 3 to 14. You can change this range by specifying different high and low values in the PROC2_SET_PRIORITY system call. These values must be between of 1 to 16. The PPRI (process_priority) Shell command uses the PROC2_SET_PRIORITY system call.

Note that the Display Manager (DM) process has a priority range of 16, 16. See Section 4.5 for an example of PROC2_SET_PRIORITY in a program.

4.4. Assigning a Name to a Process

When the operating system creates a process, it assigns a number or unique identifier (UID) to the process. You might want to refer to the process by name rather than number so that you can identify it more easily. You do so with the PM_SET_NAME system call. You supply the name, its length, and process UID; the call returns a completion status.

After assigning a name to a process with PM_SET_NAME, Shell commands, such as PPRI (process_priority), PST (process_status), and SIGP (signal_process) refer to the process by its name.

Server programs such as "alarm_server" or "mail" use PM_SET_NAME to name their server processes. Users can name a process they create by using the -N option with the Shell command CRP (create_a_process), and the DM commands, CP (create_process), CPO (create_process_only), and CPS (create_process_server).

NOTE: A process can be assigned a name only once. You cannot rename or remove a name once it is assigned.

If you try to name a process that already has a name, you will get the error, PM_ALREADY_NAMED.

4.5. Sample Programs Using PM, PROC2 System Calls

Example 4-3 is a sample program using PM_SET_NAME to assign a name to a process, and PROC2_SET_PRIORITY to change its priority. The program first creates an unnamed process using PGM_INVOKE, and then sets the name. Before it sets the name, it must first get its process UID using the PGM_GET_PUID system call.
PROGRAM pm_proc2_set_name_priority;

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/proc2.ins.pas';
%include '/sys/ins.pm.ins.pas';
%include '/sys/ins/streams.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';
%list;

CONST
  lo_bound = 9;  \{ Low bound of process priority range \}
  hi_bound = 5;  \{ High bound of process priority range \}

VAR
  uid : uid$_t$;  \{ Process UID \}
  status : status$_t$;
  conv : pgm$_connv$;  \{ Connection vector \}
  handle : pgm$_proc$;  \{ Process handle \}
  prog_name : name$_name$_t := 'invoke_clock';  \{ Name to give process \}

PROCEDURE check_status;  \{=================================== \}
BEGIN 
  IF status.all <> status$_ok$ THEN
    BEGIN
      error$_print$ (status);
      pgm$_exit$;
    END;
  END; { check_status }
BEGIN  \{ Main ............................................... \}
  writeln;
  writeln ( 'This program invokes a process to run a digital clock. ');
  writeln ( 'names the process, INVOKE_CLOCK, and sets its priority. ');
  writeln ( 'You can see the process listed by name by invoking the ');
  writeln ( 'Shell command PST (process_status). ');
  writeln;

  \{ Load the standard streams to pass to the invoking program. \}
  conv[0] := stream$_stdin$;
  conv[1] := stream$_stdout$;

  \{ Create an unnamed child process using PGM$_INVOKE$. \}

Example 4-3. Setting Name and Priority of a Process

Apollo Confidential  4-7  Managing Programs
Example 4-3. Setting Name and Priority of a Process
Chapter 5
Handling Dynamic Storage

The DOMAIN system contains two managers that allocate dynamic storage. The Read/Write Storage Manager (RWS) is a set of DOMAIN system routines that allocates dynamic storage. These system calls allow you to request a specific amount of storage during the execution of your program. Each call returns a pointer to the address of the new storage space.

The Basic Allocate/Free Heap Manager (BAF) allows you to create a heap and perform allocate and free operations on that heap. You would use the BAF manager when you want to specify where the storage comes from, or have a stricter control over allocating and releasing storage.

We describe both the RWS and BAF managers in this chapter.

5.1. System Calls, Insert Files, and Data Types

To use the RWS manager, use the system calls with the prefix RWS. To use the BAF manager, use the system calls with the prefix BAF. This chapter describes how most of these calls work. For details on system call syntax, data types, and error messages, see Part II of this manual. See also the DOMAIN System Call Reference manual for details on previously released RWS calls.

When using RWS or BAF system calls in your program, you must specify the appropriate insert file for the language your are using. Where prefix is the desired subsystem (RWS for read/write storage, BAF for basic allocate/free heap storage), the insert files are RWS insert files are

/SYS/INS/prefix.INS.C for C.
/SYS/INS/prefix.INS.FTN for FORTRAN.
/SYS/INS/prefix RWS.INS.PAS for Pascal.

5.2. Overview of the RWS System Calls

RWS system calls provide a few different ways to allocate storage. Which method you should use depends on

- How long you want to keep the storage.
- How much system overhead you can afford.
- How you want to the storage to be accessed, for example, within the calling process or among all processes.

Table 5-1 lists the RWS system calls you can use to allocate storage dynamically. Details on each type of storage follow.
### Table 5-1: RWS System Calls to Allocate Dynamic Storage

<table>
<thead>
<tr>
<th>System Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWS_$ALLOC</td>
<td>Allocates read/write storage for FORTRAN or Pascal programs.</td>
</tr>
<tr>
<td>RWS_$ALLOC_HEAP</td>
<td>Allocates heap* (releasable read/write) storage for programs.</td>
</tr>
<tr>
<td>RWS_$ALLOC_HEAP_POOL</td>
<td>Allocates heap* (releasable read/write) storage in a specified pool.</td>
</tr>
<tr>
<td>RWS_$ALLOC_RW</td>
<td>Allocates read/write storage for Pascal programs only.</td>
</tr>
<tr>
<td>RWS_$ALLOC_RW_POOL</td>
<td>Allocates read/write storage in a specified pool.</td>
</tr>
<tr>
<td>RWS_$RELEASE_HEAP</td>
<td>Releases the storage you allocated with RWS_$ALLOC_HEAP or RWS_$ALLOC_HEAP_POOL.</td>
</tr>
</tbody>
</table>

* When these calls allocate read/write storage, they provide you with pointers to storage that can be released when you no longer need it. To distinguish this from unreleasable read/write storage, we refer to this as heap storage. Note that this differs from a BAF heap described in Section 5.3.

You can use all the above systems calls in Pascal and C programs. However, in C programs, you might want to use the C Library routine, MALLOC. Note that due to FORTRAN calling conventions, the only RWS call you can use in FORTRAN programs is RWS_$ALLOC. This limitation will be corrected in a future AEGIS Software Release.

Whether you allocate read/write storage or heap storage depends on how long you want to keep the storage. Once you allocate read/write storage, the storage exists until the program terminates. However, you can explicitly release heap storage once you have finished using it.

The heap requires more system overhead initially. Currently, an allocation from the RWS heap requires between 4 to 16 bytes of overhead — to keep track of the allocated storage. Note that the amount of overhead is subject to change, so your program should not depend on an exact amount of system overhead. The system requires no overhead to allocate read/write storage.

You usually want to allocate heap storage during your program if you need a substantial amount of storage for a limited period of time, or if you want to keep your working set as small as possible.

For example, the CL allocates heap storage to hold the tokens while parsing the command line. It frees the storage after parsing the line. The CL uses heap storage because it can tell when it no longer needs the storage. Also, if it doesn’t free the storage, it would eventually run out.

You would allocate read/write storage if the amount of overhead for a heap is unacceptable, or if you do not need to release the storage before terminating the program.
If you want to control how the dynamic storage can be accessed, specify the appropriate option in the RWS_$POOL system calls. You can allocate both read/write storage and heap storage in these pools so that you can

- Limit storage to local access within the calling process. Specify the **standard pool** option (RWS_$STD_POOL) in most cases.

- Make storage accessible to all processes. Specify the **global pool** option (RWS_$GLOBAL_POOL) to share information among processes. For example, you might want to implement a global queue to pass messages between processes. Note that pointers are valid in all processes because all processes have a reserved identical portion of address space.

- Make storage accessible to the calling process and to an overlay process. Specify the **stream pool** option (RWS_$STREAM_TM_POOL) when your program needs to pass information to a program invoked with a UNIX EXEC system call. For example, the STREAM manager uses RWS_$STREAM_TM_POOL to pass an open stream to a program invoked with an EXEC call. It stores information about that stream in the stream pool.

Table 5-2 summarizes the aspects of each type of storage allocation.

**Table 5-2. Summary of Types of Storage Allocation**

<table>
<thead>
<tr>
<th>Read/Write Storage</th>
<th>Heap Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Pool</strong></td>
<td><strong>Storage kept until program exits or until it invokes a program with a UNIX EXEC system call.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>No system overhead.</strong></td>
</tr>
<tr>
<td><strong>Global Pool</strong></td>
<td><strong>Storage kept until reboot.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>About 4 bytes of system overhead.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Storage available to all processes.</strong></td>
</tr>
</tbody>
</table>
Table 5-2. Summary of Types of Storage Allocation, Cont.

<table>
<thead>
<tr>
<th>Read/Write Storage</th>
<th>Heap Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Storage kept until</td>
<td>Storage kept until you release it with RWS_$RELEASE_HEAP.</td>
</tr>
<tr>
<td>Pool program exits.</td>
<td>About 16 bytes of system overhead.</td>
</tr>
<tr>
<td>No system overhead.</td>
<td>Storage available to the local process or to a program invoked with a UNIX EXEC system call.</td>
</tr>
</tbody>
</table>

NOTE: Do not depend on the exact amount of system overhead used in RWS system calls. The amount of overhead is subject to change.

5.3. Overview of the BAF System Calls

The basic allocate/free (BAF) heap manager allows you to create a heap to handle dynamic storage allocation yourself. The advantage to handling storage with the BAF manger is that you can specify from where BAF gets the storage. You also have tighter control over allocating and freeing the storage you use.

For example, you could allocate storage with BAF system calls to control the size of your working set by subdividing your heap into smaller heaps. Place storage referenced closer in time in the same heap.

To use the BAF manager, you follow this sequence:

1. Create a heap with BAF_$CREATE.
2. Allocate storage from the heap with BAF_$ALLOC.
3. Release the storage and return it to the heap with BAF_$FREE when your program is through using the storage.
4. Add storage space to the heap, if necessary, with BAF_$ADD_SPACE.

When creating the heap, you can specify the storage space in many ways, for example, declaring an array in a program, using an RWS_$ALLOC_RW or MS_$CRMAPL system call.

If the storage is shared, you must specify the BAF_$SHARED option in the first parameter of BAF_$CREATE. By setting the BAF_$SHARED option, subsequent calls to BAF_$ALLOC and BAF_$FREE will obtain locks before performing their operation. This is necessary to prevent two processes from corrupting the heap by using it at the same time.
Once you have created the heap, you can perform any number of allocate and free operations. Use BAF_$ALLOC to allocate storage from the heap. BAF_$ALLOC allocates a contiguous region of storage up to a maximum of 32K bytes. If you require more storage, you can make subsequent calls to BAF_$ALLOC.

BAF_$ALLOC could fail if there’s not enough storage space left in the heap (it returns the error BAF_$NO_ROOM). You can add space to the heap with BAF_$ADD_SPACE.

BAF_$ALLOC and BAF_$FREE could fail if the heap overwrites the overhead information (pointers, size) which is located before each block. This can occur if your program refers to an array beyond its bounds.

5.3.1. Improving Performance Under Current Implementation

This section describes some characteristics of the BAF manager that might help you improve performance. We reserve the right, however, to change details of this implementation in the future to improve efficiency.

The following implementation details might help you use a BAF heap more efficiently:

- The current implementation of BAF manager uses a single tag before the block of storage to keep track of the block.

- BAF_$ALLOC combines adjacent free items while searching for the free list for a large enough block to perform the allocate operation.

- When you free items, BAF_$FREE puts the free blocks on a free list. Therefore, freeing is very efficient.

- Allocating and freeing blocks of the same size is very fast.

- The BAF manager tends to reuse recently freed blocks, so you get desirable working set behavior.

5.4. Sample Program Using RWS and BAF System Calls

The program in Example 5-1 shows how to use BAF system calls to allocate storage. It uses the MS_CRMPL system call to create a temporary file to store the heap. (You might want to use a file to maintain storage at different program levels in a single process.) The program then performs allocates and frees from the file using BAF system calls.
PROGRAM manage_mapl_heap;

CONST
  heap_chunk_size = 32 * 1024;

TYPE
  stack_pointer = "stack_item;
  stack_item = RECORD
    stack_data : string;
    next_link : stack_pointer;
  END;

VAR
  stack : univ_ptr;
  amount : integer;
  command : string;
  command_len : integer;
  name : string;
  new_name : string;
  name_len : integer;
  address : stack_pointer := NIL;
  old_top : stack_pointer;
  null_line : boolean;
  stack_ptr : stack_pointer := NIL;
  stack_address : baf_$t;
  status : status_$t;
  ok : boolean;

{ Exit on error procedure  _______________________________ }

PROCEDURE check_status (IN status : UNIV status_$t);
BEGIN
  IF status.all <> status_$ok THEN
    BEGIN
      error_$std_format(status, '%%s');
      pgm_$set_severity(pgm_$error);
      pgm_$exit;
    END;
  END; { err_exit }

Example 5-1. Allocating Storage with BAF System Calls
{ Allocate storage for new top of stack =============================== }

PROCEDURE add_storage ( IN new_name : string;
    IN OUT top : stack_pointer );

VAR
    temp_ptr : stack_pointer;

BEGIN { add_storage }

    { Allocate enough storage for the new name. }
    temp_ptr := BAF_ALLOC (stack_address, { Address of heap from
        sizeof (stack_item). { Size of storage to allocate }
    status ); { Completion status }

    check_status(status);

    IF temp_ptr = NIL THEN
        BEGIN
            { Create another temporary file to get more storage. }
            stack := MS_CRMAPL ( { Object to be mapped }
                0, { Length of object }
                0, { First byte to be mapped }
                heap_chunk_size, { Number of bytes to map }
                ms$nr_xor_lw, { Allow 1 writer, any readers }
                status ); { Completion status }

            check_status (status);

            BAF_ADD_SPACE ( stack_address, { Address of heap to which to add
                stack, { Address of added storage }
                heap_chunk_size, { Amount of storage to add }
                status ); { Completion status }

            check_status(status);

            temp_ptr := BAF_ALLOC (stack_address, { Address of heap from which
                sizeof(stack_item). { Size of storage to allocate }
                status ); { Completion status }

            check_status(status);

            temp_ptr$.stack_data := new_name; { Link new name }
            temp_ptr$.next_link := top; { Connect to previous top }
            top := temp_ptr; { Redefine top of stack }

        END;

END; { add_storage }

Example 5-1. Allocating Storage with BAF System Calls, Cont.
FUNCTION empty_stack (IN top : stack_pointer) : boolean;
BEGIN  { empty_stack }
    empty_stack := top = NIL
END;  { empty_stack }

PROCEDURE free_storage ( OUT del_name : string;
                          IN OUT top : stack_pointer );
BEGIN  { free_storage }
    IF empty_stack(top) THEN
        writeln (' Cannot remove name from empty list. ')
    ELSE BEGIN
        del_name := top^.stack_data;  { Get name to delete }
        old_top := top;               { Save old top of stack }
        top   := top^.next_link;      { Make next name top of stack }

        baf_free ( stack_address,  { Address of storage from which to free
                                   block }
                   old_top,    { Address of block of storage to be freed }
                   status );  { Completion status }
        check_status(status);
    END;
END;  { free_storage }

BEGIN  { Main }

{ Allocate file to hold stack info then create a heap to handle the stack. }

stack := MS_$CRMAPL ( '',  { Object to be mapped }
                       0,   { Length of object }
                       0,   { First byte to be mapped }
                       heap_chunk_size,   { Number of bytes to map }
                       ms_$nr_xor_1w,    { Allow 1 writer, any readers }
                       status );        { Completion status }

check_status(status);

IF stack <> NIL THEN
BEGIN
    stack_address := BAF_CREAT ( [],  { No options }
                                 stack,  { Address of storage to use }
                                 heap_chunk_size,   { Maximum size of stack }
                                 status );  { Completion status }
    check_status(status);
END;

Example 5-1. Allocating Storage with BAF System Calls, Cont.
Example 5-1. Allocating Storage with BAF System Calls, Cont.
Part II. DOMAIN Advanced System Call Reference
Introduction

This part of Programming with DOMAIN Advanced System Calls is the Reference Section. It describes the data types, call syntax, and error messages for the programming calls described in Part I of this book. For your convenience, we structured this part like the DOMAIN System Call Reference manual: The sections are divided by system manager, arranged alphabetically by system manager name.

If you prefer, you can insert these pages in the proper place within the reference manual. In cases where this book documents additional system calls to existing subsystems (for example, RWS) we duplicated the entire data types and error sections of the subsystem, marking the new material with change bars.

The rest of this introduction describes the DOMAIN system insert files and the format of the information found in the sections that follow.

DOMAIN Insert Files

The DOMAIN system provides insert files that define data types, constants, values, and routine declarations. The insert files also define the exact form of each system call or routine. (Even the FORTRAN version does this using comments, although the FORTRAN compiler doesn't check the forms that you use.)

To use system calls of a particular subsystem, you must specify the include file for that subsystem in your program. All insert files are located in the directory /SYS/INS/. For example, if you are using system error routines in a Pascal program, you include the insert file, /SYS/INS/ERROR.INS.PAS. Using the same routines in a FORTRAN program, you include /SYS/INS/ERROR.INS.FTN. All insert files are specified using the syntax

/SYS/INS/subsystem-prefix.INS.language-abbreviation

where the language abbreviation is PAS (Pascal), FTN (FORTRAN), or C (C). The listing on the next page shows all the available insert files.

In addition to including required subsystem insert files in a program, you must always include the BASE insert file for your programming language. These files contain some basic definitions that a number of subsystem routines use.

You specify BASE insert files using the syntax

/SYS/INS/BASE.INS.language-abbreviation
## Summary of Insert Files

<table>
<thead>
<tr>
<th>Insert File</th>
<th>Operating System Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>/SYS/INS/BASE.INS.lan</td>
<td>Base definitions -- must always be included</td>
</tr>
<tr>
<td>/SYS/INS/ACLM.INS.lan</td>
<td>Access Control List manager</td>
</tr>
<tr>
<td>/SYS/INS/BAF.INS.lan</td>
<td>Basic Allocate and Free manager</td>
</tr>
<tr>
<td>/SYS/INS/CAL.INS.lan</td>
<td>Calendar manager</td>
</tr>
<tr>
<td>/SYS/INS/CL.INS.lan</td>
<td>Command Line Handler</td>
</tr>
<tr>
<td>/SYS/INS/ERROR.INS.lan</td>
<td>Error reporting manager</td>
</tr>
<tr>
<td>/SYS/INS/EC2.INS.lan</td>
<td>Eventcount manager</td>
</tr>
<tr>
<td>/SYS/INS/FEU.INS.lan</td>
<td>File and Tree Utility</td>
</tr>
<tr>
<td>/SYS/INS/GM.INS.lan</td>
<td>Graphics Metafile Resource</td>
</tr>
<tr>
<td>/SYS/INS/GMF.INS.lan</td>
<td>Graphics Map Files manager</td>
</tr>
<tr>
<td>/SYS/INS/GFR.INS.lan</td>
<td>Graphics Primitives manager</td>
</tr>
<tr>
<td>/SYS/INS/IPC.INS.lan</td>
<td>Interprocess Communications datagrams</td>
</tr>
<tr>
<td>/SYS/INS/KBD.INS.lan</td>
<td>[Useful constants for keyboard keys]</td>
</tr>
<tr>
<td>/SYS/INS/LOADER.INS.lan</td>
<td>Object module loader</td>
</tr>
<tr>
<td>/SYS/INS/LOGIN.INS.lan</td>
<td>Login manager</td>
</tr>
<tr>
<td>/SYS/INS/MBX.INS.lan</td>
<td>Mailbox manager</td>
</tr>
<tr>
<td>/SYS/INS/MS.INS.lan</td>
<td>Mapping server</td>
</tr>
<tr>
<td>/SYS/INS/MTS.INS.lan</td>
<td>Magtape/streams interface</td>
</tr>
<tr>
<td>/SYS/INS/MUTEX.INS.lan</td>
<td>Mutual exclusion lock manager</td>
</tr>
<tr>
<td>/SYS/INS/NAMES.INS.lan</td>
<td>Naming server</td>
</tr>
<tr>
<td>/SYS/INS/PAD.INS.lan</td>
<td>Display manager</td>
</tr>
<tr>
<td>/SYS/INS/PBUF.INS.lan</td>
<td>Paste buffer manager</td>
</tr>
<tr>
<td>/SYS/INS/PFM.INS.lan</td>
<td>Process fault manager</td>
</tr>
<tr>
<td>/SYS/INS/PGM.INS.lan</td>
<td>Program manager</td>
</tr>
<tr>
<td>/SYS/INS/PM.INS.lan</td>
<td>User process routines</td>
</tr>
<tr>
<td>/SYS/INS/PROC1.INS.PAS</td>
<td>Process manager (Pascal only)</td>
</tr>
<tr>
<td>/SYS/INS/PROC2.INS.lan</td>
<td>User process manager</td>
</tr>
<tr>
<td>/SYS/INS/RWS.INS.lan</td>
<td>Read/write storage manager</td>
</tr>
<tr>
<td>/SYS/INS/SIO.INS.lan</td>
<td>Serial I/O interface</td>
</tr>
<tr>
<td>/SYS/INS/SMDU.INS.lan</td>
<td>Display driver</td>
</tr>
<tr>
<td>/SYS/INS/STREAMS.INS.lan</td>
<td>Stream manager</td>
</tr>
<tr>
<td>/SYS/INS/TIME.INS.lan</td>
<td>Time manager</td>
</tr>
<tr>
<td>/SYS/INS/TONE.1an</td>
<td>Speaker manager</td>
</tr>
<tr>
<td>/SYS/INS/TPAD.INS.lan</td>
<td>Touchpad manager</td>
</tr>
<tr>
<td>/SYS/INS/VEC.INS.lan</td>
<td>Vector arithmetic routines</td>
</tr>
<tr>
<td>/SYS/INS/VFMT.INS.lan</td>
<td>Variable formatter</td>
</tr>
</tbody>
</table>
Organizational Information

This introductory section is followed by sections for each subsystem. The material for each subsystem is organized into the following three parts:

1. Detailed data type information (including illustrations of records for the use of FORTRAN programmers).

2. Full descriptions of each system call. Each call within a subsystem is ordered alphabetically.

3. List of possible error messages.

Data Type Sections

A subsystem’s data type section precedes the subsystem’s individual call descriptions. Each data type section describes the predefined constants and data types for a subsystem. These descriptions include an atomic data type translation (i.e., TIME_$REL_ABS_T = 4-byte integer) for use by FORTRAN programmers, as well as a brief description of the type’s purpose. Where applicable, any predefined values associated with the type are listed and described. The following is an example of a data type description for the RWS_$POOL_T type.

DATA TYPES

RWS_$POOL_T

A 2-byte integer. Types of pools to allocate read/write or heap storage. One of the following pre-defined values:

RWS_$STD_POOL
Standard pool makes storage accessible to calling process only.

RWS_$STREAM_TM_POOL
Stream pool makes storage accessible to calling program and to a program invoked with the UNIX EXEC system call.

RWS_$GLOBAL_POOL
Global pool makes storage accessible to all processes.

In addition, the record data types are illustrated in detail. Primarily, we have geared these illustrations to FORTRAN programmers who need to construct record-like structures, but we’ve designed the illustrations to convey as much information as possible for all programmers.
Each record type illustration does the following:

- Clearly shows FORTRAN programmers the structure of the record that they must construct using standard FORTRAN data type statements. The illustrations show the size and type of each field.

- Describes the fields that make up the record.

- Lists the byte offsets for each field. These offsets are used to access fields individually.

- Indicates whether any fields of the record are, in turn, predefined records.

The following is the description and illustration of the PM$_{SECT}$ _INFO predefined record:

<table>
<thead>
<tr>
<th>Total Size: 40</th>
<th>byte: offset</th>
<th>field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0:</td>
<td>char</td>
<td>name</td>
</tr>
<tr>
<td>Predefined Record:</td>
<td>32:</td>
<td>integer</td>
</tr>
<tr>
<td>PM$_{SECT}$ _INFO</td>
<td>36:</td>
<td>integer</td>
</tr>
</tbody>
</table>

NAME
The name of the section, a character array of up to 32 elements.

LOC
A 4-byte integer. Location of section information.

LEN
A 4-byte integer. Length of section.

FORTRAN programmers, note that a Pascal variant record is a record structure that may be interpreted differently depending on usage. In the case of variant records, as many illustrations will appear as are necessary to show the number of interpretations.
System Call Descriptions

We have listed the system call descriptions alphabetically for quick reference. Each system call description contains:

• An abstract of the call's function.
• The order of call parameters.
• A brief description of each parameter.
• A description of the call’s function and use.

These descriptions are standardized to make referencing the material as quick as possible.

Each parameter description begins with a phrase describing the parameter. If the parameter can be declared using a predefined data type, the descriptive phrase is followed by the phrase "in XXX format" where XXX is the predefined data type. Pascal or C programmers, look for this phrase to determine how to declare a parameter.

FORTRAN programmers, use the second sentence of each parameter description for the same purpose. The second sentence describes the data type in atomic terms that you can use, such as "This is a 2-byte integer." In complex cases, FORTRAN programmers are referenced to the respective subsystem’s data type section.

The rest of a parameter description describes the use of the parameter and the values it may hold.

The following is an example of a parameter description:

heap_ptr
Address of the created heap, in BAF_$T format. This is a 4-byte integer. A returned address of zero (NIL) means that BAF_$CREATE could not create a heap.

Error Sections

Each error section lists the status codes that may be returned by subsystem calls. The following information appears for each error:

• Predefined constant for the status code.
• Text associated with the error.
**BAF DATA TYPES**

**DATA TYPES**

**BAF__$OPTS_T**

A 4-byte integer. A set of BAF__$SHARED types. Indicates that the storage supplied to BAF is shared.

**BAF__$T**

A record of UNIV pointers which point to internal data structures. The diagram below illustrates the BAF__$T data type:

![Diagram](BAF__$T_diagram.png)

Field Description:

- **REP**
  - Pointer to BAF internal data structures.

**STATUS__$T**

A status code. The diagram below illustrates the STATUS__$T data type:

![Diagram](STATUS__$T_diagram.png)

Field Description:

- **ALL**
  - All 32 bits in the status code.

- **FAIL**
  - The fail bit. If this bit is set, the error was
within the scope of the module invoked, but occurred within a lower-level module (bit 31).

**SUBSYS**
The subsystem that encountered the error (bits 24 - 30).

**MODC**
The module that encountered the error (bits 16 - 23).

**CODE**
A signed number that identifies the type of error that occurred (bits 0 - 15).

**UNIV_PTR**
A 4-byte integer. A pointer to allocated storage.
BAF$_$ADD_SPACE

BAF$_$ADD_SPACE

Add more storage space to a basic allocate/free (BAF) heap that was previously created with BAF$_$CREATE.

FORMAT

BAF$_$ADD_SPACE (heap, area, size, status)

INPUT PARAMETERS

heap
The address of the heap to which the storage will be added, in BAF$_$T format. This address is the return value of the function, BAF$_$CREATE.

area
The address of the storage space to be added to the heap, in UNIV_PTR format. This is a 4-byte integer.

size
The amount of space to be added. This is a 4-byte integer. The maximum size you can add at one time is 32K bytes. If you want to add more than 32K bytes, add the space in a separate call to BAF$_$ADD_SPACE.

OUTPUT PARAMETERS

status
Completion status, in STATUS$_$T format. This data type is 4 bytes long. See the BAF Data Types for more information.

USAGE

Use this call to add storage to a basic alloc/free heap that you previously created with BAF$_$CREATE. You can add up to 32K bytes per call. If you want to add more space to the heap, make another call to BAF$_$ADD_SPACE.
BAF\_ALLOC

Allocates a contiguous region of storage from a basic allocate/free (BAF) heap.

**FORMAT**

pointer := BAF\_ALLOC (heap, size, status)

**RETURN VALUE**

pointer

Address of the allocated storage space, in UNIV\_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that BAF\_ALLOC could not allocate the desired storage from the specified heap.

**INPUT PARAMETERS**

heap

Address of the heap from which storage will be allocated, in BAF\_T format. This address is the return value of the function, BAF\_CREATE.

size

Size of the new storage space to be allocated from the heap. This is a 4-byte integer. The maximum amount of space you can allocate at one time is 32K bytes.

**OUTPUT PARAMETERS**

status

Completion status, in STATUS\_T format. This data type is 4 bytes long. See the BAF Data Types section for more information. STATUS\_T will always return STATUS\_OK unless BAF\_ALLOC returns a value of NIL in the return parameter, "pointer."

**USAGE**

Use this call to allocate storage from the basic alloc/free heap that you created with BAF\_CREATE. This call allocates a contiguous region of storage of "size" bytes long. The storage space will be aligned on a 4-byte boundary. The overhead per allocation is currently 4 bytes long, but is subject to change.
BAF\_\$CREATE

**FORMAT**

\[
\text{heap}_\text{ptr} := \text{BAF}\_\$\text{CREATE} (\text{options, area, size, status})
\]

**RETURN VALUE**

\[
\text{heap}_\text{ptr}
\]

Address of the created heap, in BAF\_\$T format. This is a 4-byte integer. A returned address of zero (NIL) means that BAF\_\$CREATE could not create a heap.

**INPUT PARAMETERS**

**options**

BAF options in BAF\_\$OPS\_T format. This is a 4-byte integer. Specify the predefined value:

**BAF\_\$\text{SHARED}**

Indicates that the storage space you supply to BAF is shared. Therefore, BAF\_\$\text{ALLOC} and BAF\_\$\text{FREE} must obtain a lock during their operations to prevent another process from corrupting the heap. For greater efficiency, specify BAF\_\$\text{SHARED} only when the storage is shared.

**area**

Address of storage space that BAF\_\$CREATE will use to create a heap in UNIV\_PTR format. This is a 4-byte integer.

**size**

Size of the heap. This is a 4-byte integer.

**OUTPUT PARAMETERS**

**status**

Completion status, in STATUS\_\$T format. This data type is 4 bytes long. STATUS\_\$T will always return STATUS\_\$\text{OK} unless BAF\_\$\text{CREATE} returns a value of NIL in the return parameter, "heap\_ptr."

See the BAF Data Types section for more information.

**USAGE**

Use this call to create a basic alloc/free heap. You can use subsequent BAF system calls to allocate storage space from the heap (with BAF\_\$\text{ALLOC}), free storage space (with BAF\_\$\text{FREE}), and add more storage space to the heap (with BAF\_\$\text{ADD\_SPACE}).
BAF_$FREE

Returns a region of storage to a basic allocate/free (BAF) heap that was previously allocated with BAF_$ALLOC.

FORMAT

BAF_$FREE (heap, block, status)

INPUT PARAMETERS

heap
The address of the heap from which storage will be freed, in BAF_$T format. This address is return value of the function BAF_$CREATE.

block
The address of the block of storage to be freed. This address is the return value of the function, BAF_$ALLOC. This is a 4-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. See the BAF Data Types section for more information. This data type is 4 bytes long. STATUS_$T will always return STATUS_$OK unless the specified block is not located in the specified heap.

USAGE

Use this call to free storage from the basic alloc/free heap that you previously allocated with BAF_$ALLOC. This call fails if the block was not located in the heap, or if it was already free.
BAF ERRORS

ERRORS

BAF_$FREED_TWICE
BAF_$FREE returns this error if you attempted to free a block that was previously freed.

BAF_$NO_ROOM
BAF_$ALLOC returns this error if there was not enough storage in the heap to allocate the blocks you specified.

BAF_$FORMAT_VIOLATED
This error occurs if the block overhead gets overwritten. It gets overwritten when another process overflows the bounds of an array, because the overhead is located just before each block.
CL DATA TYPES

DATA TYPES

CL_$OPT_T

A 2-byte integer. Set of options that control how the CL reads the command line. List some of the following predefined values. (Note that some of these options are mutually exclusive; see the CL call syntax for details.)

CL_$WILDCARDS
Causes CL_$GET_NAME to resolve wildcards and return the resolved names (default).

CL_$NO_WILDCARDS
Causes CL_$GET_NAME to return wildcard names verbatim, it does not resolve them.

CL_$VERIFY_NONE
CL_$GET_NAME does not verify any names with the user. (default).

CL_$VERIFY_ALL
CL_$GET_NAME verifies all names with the user.

CL_$VERIFY_WILD
CL_$GET_NAME verifies only names resolved by wildcards with the user.

CL_$DASH_NOP
Causes CL to treat the hyphen as a name. Normally, the hyphen is an identifier for the standard input stream (default).

CL_$DASH_NAMES
Causes CL to read names from standard input if no arguments appear on the command line. (Obsoletes for new program development.)

CL_$DASH_DFT_NOP
Suppresses any special action when there are no arguments on the command line (default).

CL_$NAME_DFT_STDIN
Causes CL to read names from standard input if no arguments appear on the command line.

CL_$NO_MATCH_OK
 Displays no warning or error message when a wildcard does not match existing pathnames.

CL_$NO MATCH WARNING
Displays a warning message on error output.
when a wildcard does not match existing pathnames (default).

**CL__NO_MATCH_ERROR**
Displays an error message on error output and terminates the program when a wildcard does not match an existing pathname.

**CL__KEYWORD_DELIM**
Prevents **CL__GET_ARG** and **CL__GET_NAME** from returning an "unused" keyword as an argument. The calls return FALSE if they encounter a keyword. (default).

**CL__NO_KEYWORD_DELIM**
Causes no special treatment of unread keywords by **CL__GET_ARG** and **CL__GET_NAME**. CL procedures return the keywords as if they were arguments.

**CL__COMMENTS**
Causes CL to ignore all characters enclosed in brackets when reading a names-file. Do not use this option for commands that accept derived names.

**CL__NO_COMMENTS**
Causes no special treatment of characters enclosed in brackets (default).

**CL__STAR_NAMES**
Allows the user to specify names-files on the command line for CL to resolve (default).

**CL__NO STAR NAMES**
Does not allow the user to specify names-files on the command line; CL treats the "**" like any other character.

**CL__OPT_SET_T**
A 4-byte integer. A set of CL options in **CL__OPT_T** format. For a list of options, see **CL__OPT_T** above.
**CL DATA TYPES**

**CL_$ARG_SELECT_T**

A 2-byte integer. Determines whether CL begins a search at the current token pointer or at the top of the token list. One of the following pre-defined values:

- **CL_$FIRST**
  - Search begins before the first unused token on the token list.

- **CL_$NEXT**
  - Search begins at the current token pointer on the token list.

**CL_$ANSWER_T**

A 2-byte integer. The answer that the user supplies when CL performs a query. One of the following pre-defined values:

- **CL_$YES**
  - User confirms pathname.

- **CL_$NO**
  - User cancels pathname, so CL ignores it.

- **CL_$QUIT**
  - User tells CL to verify any names.

- **CL_$GO**
  - User tells CL to continue processing names without verifying them.

**CL_$REQUIRED_T**

A 2-byte integer. Determines whether the user needs to supply a derived name after a keyword in the CL_$GET_FLAGGED_DERIVED_NAME call. One of the following pre-defined values:

- **CL_$REQUIRED**
  - Requires that the user supplies a derived name after the specified flag. If absent, the program terminates.

- **CL_$OPTIONAL**
  - User has the option to specify a derived name after a specified keyword. If the user supplies a derived name, it returns TRUE, and the derived name. If the user does not supply a derived name, it returns TRUE, but with no name.
CL DATA TYPES

CL_$WILD_T

A 2-byte integer. Set of wildcard options that determine how the wildcard manager resolves wildcards. One of the following pre-defined values:

CL_$WILD_FILES
Returns the names of files.

CL_$WILD_DIRS
Returns the names of directories.

CL_$WILD_LINKS
Returns the names of links.

CL_$WILD_EXCLUSIVE
Traverses each branch of the naming tree as far as the first wildcard match. This is useful for commands that operate on entire directories, such as COPY_TREE, or WRITE_BACKUP.

CL_$WILDCHASE_LINKS
Chases links that point to directories.

CL_$WILD_FIRST
Stops at the first match of a given wildcard, rather than resolving all names.

CL_$WILD_SET_T

A 2-byte integer. A set of wildcard options in CL_$WILD_T format. For a list of options see CL_$WILD_T above.

CL_$ARGV

An argument returned by CL_$PARSE_ARGS. The diagram below illustrates the CL_$ARGV data type:

The diagram below illustrates the CL_$ARGV data type:

Field Description:

LEN
Length of the argument.
CL DATA TYPES

CHARS
The text of the argument, a character array of up to 128 elements.

CL_ATTR_SET_T
A 2-byte integer. An element set, 0..15, based on the contents of two character strings.
CL_$CHECK_FLAG

Checks the command line to see if the user specified a flag. It counts the number of tokens following it that are not flags.

FORMAT
found := CL_$CHECK_FLAG(flag_string, tokens_expected)

RETURN VALUE
found
Returns TRUE if flag_string is found; returns FALSE if not.

INPUT PARAMETERS
flag_string
The flag (a token preceded by a hyphen) you want the CL to search for. This is a lowercase character string in the UNN character string format. You specify the flag in the form of 
"-required[optional]."

"-required" represents the characters of the flag that the user must type. "[optional]" represents those characters that are optional. If the user specifies any optional characters, they all must be specified.

If you do not specify any optional characters, you must supply an empty bracket, [].

For example, "-arg[uments]" matches "-arg" or "-arguments." "-ld[]" matches "-ld."

tokens_expected
The number of tokens that you expect to follow the specified flag. This is a 2-byte integer.
If you supply a negative number, the CL assumes that you want at least that many arguments, but will accept more.

USAGE
Use this call to check if the user supplied the correct number of options.

The call searches the token list for a specified flag and returns a Boolean indicating whether the flag was found. It also tests for the expected number of arguments following the flag.

If the user does not supply the number of tokens expected, the CL returns an error and calls PGM_$EXIT to terminate the program. If it finds the flag, the CL marks it "used," and moves the token pointer to the flag. Otherwise, the token pointer remains unchanged.

Normally, you do not need to know whether the user typed the abbreviated or full flag. However, you can find out what the user typed with the CL_$GET_FLAG_INFO system call.

NOTE: Do not use this call if tokens can go anywhere on a command line because you might get unexpected results. In this case, use CL_$GET_FLAG.
CL_\$CHECK\_UNCLAIMED

CL_\$CHECK\_UNCLAIMED

Checks the token list for any unread flags. If it finds any, this call prints an error message and terminates the program.

FORMAT
CL_\$CHECK\_UNCLAIMED

USAGE

Use this call to check if the user specified any options that your program does not handle. If so, the program aborts.

Most programs use this call before calling CL_\$GET\_NAME. Do not use this call if you plan to use CL_\$GET\_FLAGGED\_DERIVED\_NAME.
CL_$GET_ARG

Gets the first or next unused argument from the token list.

FORMAT

\texttt{found := \text{CL}_\$GET\_ARG(selector, arg, arg\_len, max\_len)}

RETURN VALUE

\texttt{found}

Returns \texttt{TRUE} if it finds an argument; returns \texttt{FALSE} if not.

INPUT PARAMETERS

\texttt{selector}

Determines where the CL begins to scan the token list, in \texttt{CL\_$ARG\_SELECT\_T} format. This is a 2-byte integer.

You must specify one of the following:

\texttt{CL\_$FIRST} \hspace{1em} \text{Directs the CL to return to the first unused token on the token list to begin the search.}

\texttt{CL\_$NEXT} \hspace{1em} \text{Directs the CL to scan the token list beginning with the argument following the token pointer.}

\texttt{max\_len}

The maximum number of characters to place into \texttt{"arg."}. This is a 2-byte integer. If the argument contains too many characters, the CL prints an error message and calls PGM_$EXIT to terminate the program.

OUTPUT PARAMETERS

\texttt{arg}

Returns the argument that CL finds on the token list, in character string format.

\texttt{arg\_len}

Returns the actual length of the argument, as a 2-byte integer.

USAGE

Use this call to get arguments from the command line. You can then convert the character strings to the desired type. If you expect pathnames from the command line, use \texttt{CL\_$GET\_NAME} because \texttt{CL\_$GET\_NAME} resolves wildcards. If you expect numbers from the command line, use \texttt{CL\_$GET\_NUM}.

\texttt{CL\_$GET\_ARG} reads a token from the token list. If it finds an argument, the CL marks it used, and moves the token pointer to it. If it doesn't find any arguments, the CL sets the token pointer to top of the list, before the first token.
If the token is a flag and the default CL option CL_$KEYWORD_DELIM is in effect, the CL returns FALSE. If the token is a flag and you set the CL option CL_$NO_KEYWORD_DELIM the call returns TRUE, and the value of "arg." This is useful if you want to read tokens that begin with a hyphen but you don't want to treat them as flags.
CL_$GET_DERIVED_NAME

Gets the next derived name that applies to the name most recently read by 
CL_$GET_NAME.

FORMAT

found := CL_$GET_DERIVED_NAME(name, name_len, max_len)

RETURN VALUE

found

Returns TRUE if it finds a name; returns FALSE if not.

INPUT PARAMETERS

max_len

The maximum number of characters to place into "name." This is a 2-byte integer. If the 
name contains too many characters, the CL prints an error message and calls PGM_$EXIT 
to terminate the program.

OUTPUT PARAMETERS

name

Returns the name the CL found, in character string format. This is valid only if "found" is 
TRUE.

name_len

Returns the actual length of the name, as a 2-byte integer.

USAGE

Use this call after CL_$GET_NAME to get a derived name. Do not use in programs if 
you do not expect derived names.

Each time you call CL_$GET_NAME, the CL sets a "derived-name pointer" next to the 
token pointer. Each subsequent call to CL_$GET_DERIVED_NAME, advances the 
derived-name pointer to the next derived name, until there are none left.

A single derived name may be seen by a program a number of times. For example, the 
command line, cpf ?*.pas =.05.31, matches all instances in the working directory of files 
ending with the .pas extension. The program sees the derived name, not the token. For 
extample, if it finds the pathname foo.pas, CL_$GET_DERIVED_NAME returns 
foo.pas.05.31.

CL_$GET_DERIVED_NAME does not change the token pointer. Once a token is used 
as a derived name, it is marked "used," and is not returned by CL_$GET_NAME or 
CL_$GET_ARG again.
**CL_GET_ENUM_FLAG**

Scans the token list for any member of a list of flags that you supply. It counts the number of tokens following it which are not flags.

**FORMAT**

```plaintext
index := CL_GET_ENUM_FLAG(selector, flag_list, token_count)
```

**RETURN VALUE**

*index*

Returns the index of the flag in the "flag_list," if found; returns zero if it does not find any of the specified flags.

**INPUT PARAMETERS**

*selector*

Determines where the CL begins to scan the token list, in CL_ARG_SELECT_T format. This is a 2-byte integer.

You must specify one of the following:

- **CL\_FIRST** Directs the CL to return to the first unused token on the token list to begin the search.
- **CL\_NEXT** Directs the CL to scan the token list beginning with the argument following the token pointer.

*flag_list*

The flags you want the CL to search for. This is a lowercase character string, where each flag is in the form of "-required[optional]."

"-required" represents the characters of the flag that the user must type. "optional" represents those characters that are optional. If the user specifies any optional characters, they all must be specified.

If you do not specify any optional characters, you must supply an empty bracket, [ ]. For example, "-arg[uments]\" matches "-arg" or "-arguments." "-ld[ ]" matches "-ld".

You must separate each flag by spaces, and terminate the string with any non-space character other than a hyphen. For example, "-brief -list X".

**OUTPUT PARAMETERS**

*token_count*

Returns the number of tokens that follow the flag up to the next flag. This 2-byte integer is valid only if "index" does not equal zero.

Note that this is the number of tokens listed on the command line only. It does not count the number of names resolved from a wildcard name.
**USAGE**

Use this call when the user can supply different options that require the same program action.

The call searches the token list for specified flags from the "flag_list." If the user specifies more than one flag that is on the list, the CL returns only the first one it matches.

If it finds the flag, the CL marks it "used," and moves the token pointer to the flag. Otherwise, the token pointer remains unchanged.

Normally, you do not need to know whether the user typed the abbreviated or full flag. However, you can find out what the user typed with the CL_$_GET_FLAG_INFO call.
CL_$GET_FLAG

Checks the command line to see if the user specified a flag, which is a token preceded by a hyphen. It counts the number of tokens following it which are not flags.

FORMAT
found := CL_$GET_FLAG(flag_string, token_count)

RETURN VALUE
found
Returns TRUE if "flag_string" is found; returns FALSE if not.

INPUT PARAMETERS
flag_string
The flag you want the CL to search for. This is a lowercase character string in the form of "-required[optional]."

"-required" represents the characters of the flag that the user must type. "optional" represents those characters that are optional. If the user specifies any optional characters, they all must be specified.

If you do not specify any optional characters, you must supply an empty bracket, []

For example, "-arg[uments]" matches "-arg" or "-arguments"; "-ld[]" matches "-ld".

OUTPUT PARAMETERS
token_count
Returns the number of tokens that follow the flag up to the next flag. This 2-byte integer is valid only if the call returns TRUE.

Note that this is the number of tokens listed on the command line only. It does not count the number of names resolved from a wildcard name.

USAGE
Use this call when you want to get any specific flag from the command line. This call searches the token list for the specified flag and returns a Boolean indicating whether the flag was found.

If it finds the flag, the CL marks it "used," and moves the token pointer to the flag. If not, the token pointer remains unchanged.

This is the most commonly used call for getting flags from the command line. See also CL_$CHECK_FLAG and CL_$GET_ENUM_FLAG for other ways of getting flags. You can use CL_$GET_INFO to see which flags the user provided.
CL_$GET_FLAGGED_DERIVED_NAME

Scans the token list for a specified flag followed by a derived name. If a derived name follows the flag, it returns the name found.

FORMAT

found := CL_$GET_FLAGGED_DERIVED_NAME(flag, required_name, name, name_len, max_len)

RETURN VALUE

found

Returns TRUE if it finds a name; returns FALSE if not.

INPUT PARAMETERS

flag

The flag you want the CL to search for. This is a lowercase character string in the form of "-required[optional]."

"-required" represents the characters of the flag that the user must type. "optional" represents those characters that are optional. If the user specifies any optional characters, they all must be specified.

If you do not specify any optional characters, you must supply an empty bracket, [].

For example, "-arg[uments]" matches "-arg" or "-arguments"; "-ld[]" matches "-ld".

required_name

Specifies whether a derived name must follow the flag, in CL_$REQUIRED_T format. This is a 2-byte integer.

If you supply CL_$REQUIRED, and a derived name does not follow the flag, the CL returns an error message and terminates the program. If you supply CL_$OPTIONAL, and a derived name does not follow the flag, the CL returns TRUE, but does not return a name.

max_len

The maximum number of characters to place into "name." This is a 2-byte integer. If the name contains too many characters, the CL prints an error message and calls PGM_$EXIT to terminate the program.

OUTPUT PARAMETERS

name

Returns the derived name the CL found, in character string format. This is valid only if "found" is TRUE.

name_len

Returns the actual length of the name. This is a 2-byte integer. If CL_$OPTIONAL is specified and a derived name does not follow the flag, "name_len" is set to zero.
CL_$GET_FLAGGED_DERIVED_NAME

USAGE

Use this call after CL_$GET_NAME to check for a specific flag followed by a derived name. You can call CL_$GET_FLAGGED_DERIVED_NAME either before or after calling CL_$GET_DERIVED_NAME.

A single derived name may be seen by a program a number of times. For example, the command line, cmf {?}.04.30 @1 -r @1.rpt is handled by three calls: CL_$GET_NAME resolves the wildcard to match all instances of files in the working directory ending with the .04.30 extension. CL_$GET_DERIVED_NAME gets each derived name, which is the wildcard match without the extension. Then CL_$GET_FLAGGED_DERIVED_NAME adds the rpt extension.

CL_$GET_FLAGGED_DERIVED_NAME does not change the token pointer, or derived-name token pointer. Once a token is used as a derived name, the CL marks it "used," and does not return it to by CL_$GET_NAME or CL_$GET_ARG again.

NOTE: Do not search for this flag with CL_$GET_FLAG if you plan to use this call.
CL_$GET_FLAG_INFO

Returns a pointer to the text of the last flag found by CL_$GET_FLAG, CL_$CHECK_FLAG, CL_$GET_ENUM_FLAG.

FORMAT

CL_$GET_FLAG_INFO(flag_ptr, flag_len)

OUTPUT PARAMETERS

flag_ptr
Returns pointer to the character string containing the text of the flag. If the user did not specify a flag, it returns NIL.

flag_len
Number of characters in the character string pointed to by "flag_ptr." This is a 2-byte integer. This value is undefined if "flag_ptr" is NIL.

USAGE

Use this call to see which flag the user actually specified. You might want to return this text with an error message.
CL_$GET_NAME

CL_$GET_NAME

Gets the first or next unused name from the token list.

FORMAT

found := CL_$GET_NAME(selector, name, name_len, max_len)

RETURN VALUE

found

Returns TRUE if it finds a name; returns FALSE if not.

INPUT PARAMETERS

selector

Determines where the CL begins to scan the token list, in CL_$ARG_SELECT_T format. This is a 2-byte integer.

You must specify one of the following:

CL_$FIRST  Directs the CL to return to the first unused token on the token list to begin search.

CL_$NEXT    Directs the CL to scan the token list beginning with the argument following the token pointer.

max_len

The maximum number of characters to place into "name." This is a 2-byte integer. If the name contains too many characters, the CL prints an error message and calls PGM_$EXIT to terminate the program.

OUTPUT PARAMETERS

name

Returns the name the CL found, in character string format. This is valid only if "found" is TRUE.

name_len

Returns the actual length of the name, as a 2-byte integer.

USAGE

Use this call to get names from the command line. If it finds a name, the CL marks it "used" and moves the token pointer to it. If it doesn't find any names, the CL moves the token pointer to top of the list, before the first token.

CL_$GET_NAME reads a token from the token list. If the token is a pathname, it returns the pathname verbatim. If it is a wildcard name and the default CL_$WILDCARD option is in effect, the CL resolves the wildcard and returns the first pathname that matches. Subsequent calls to CL_$GET_NAME return successive pathnames resulting from the wildcard.
If the token is a flag and the default CL\_\$KEYWORD\_DELIM option is in effect, the call returns FALSE; therefore, preventing the call from returning an unused flag as a name.

If CL\_\$NO\_KEYWORD\_DELIM option is set and the token is a flag, the call returns TRUE, and the flag verbatim.

If you set one of the CL verify options VERIFY\_\$WILD or VERIFY\_\$ALL, this call queries the user for a yes/no approval. If the user answers "no," the CL ignores the name and processes the next name.

If the call does not find a match for a wildcard-name, CL consults the current state of the CL match option to determine whether it should terminate the program with a warning or error message, or should continue.

Normally, you do not need to know whether the token was a pathname or wildcard-name. However, you can find out exactly what the user typed with the CL\_\$GET\_INFO system call.
CL$_GET_NAME_INFO

Determines if the previous CL$_GET_NAME operated on a wildcard. If so, the call returns a pointer to the wildcard name.

FORMAT
CL$_GET_NAME_INFO(wild_ptr, wild_len)

OUTPUT PARAMETERS

wild_ptr
Returns a pointer to the character string containing the wildcard name. If the user did not specify a wildcard name, it returns NIL. This is a UNIV_PTR.

wild_len
Length of the character string pointed to by *wild_ptr.* This is a 2-byte integer. This value is undefined if "wild_ptr" is NIL.

USAGE
Use this call to determine whether the user supplied a full pathname or a wildcard name on the command line.
CL__$GET__NUM

Checks the token list for a decimal numeric argument and converts it to a 4-byte integer.

FORMAT

found := CL__$GET__NUM(number)

RETURN VALUE

found

Returns TRUE if it finds a number; returns FALSE if not.

OUTPUT PARAMETERS

number

Returns a long integer (integer32) containing the decimal numeric argument. This is a 4-byte integer.

USAGE

Use this call to convert the argument following the token pointer to a long decimal. If the argument is not a decimal number, CL__$GET__NUM prints an error message and calls PGM_$EXIT to terminate the program.

If the CL finds a number, the CL marks it "used" and moves the token pointer to it. Otherwise, the token pointer remains unchanged.
$GET SET

Builds a 16-element set from a character string.

**FORMAT**

set := CL\_\$GET\_SET(char\_string, string\_len, token, token\_len)

**RETURN VALUE**

set

Returns the set, in the CL\_\$ATTR\_SET\_T format. This is a 2-byte integer.

**INPUT PARAMETERS**

char\_string

Character string that defines the allowable token characters and their order in the returned set. This is in UNIV string format. The first character corresponds to the returned set element "0", the second element, to element "1," and so on.

string\_len

Number of characters in "char\_string." This is a 2-byte integer.

token

Token, usually read from the token list, with which to build the set. This is a UNIV character string.

token\_len

Number of characters in "token." This is a 2-byte integer.

**USAGE**

Use this call to build a set from a character string. For example, EDACL turns the character string "PGNDWRX" into a set for setting ACLs.
CL_$INIT

Initializes the command line handler that parses command lines. You must use this call before any other CL calls.

FORMAT

CL_$INIT([cl_option], program_name, program_len)

INPUT PARAMETERS

cl_option
Set of options, in CL_$OPT_SET_T format, that tells the CL how to read the command line. This is a 4-byte integer. The options control how the CL interacts with the user.

You can specify a default set of options with empty brackets, []. You need to specify only the options that are not default. The chart below lists the default CL options and their alternatives. Note that you can either take the default option, or specify one of its corresponding alternatives.

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$WILDCARDS</td>
<td>CL_$NO_WILDCARDS&lt;br&gt;Causes CL_$GET_NAME to return wildcard-names verbatim; it does not resolve them.</td>
</tr>
<tr>
<td>Causes CL_$GET_NAME to resolve wildcards and return the resolved names.</td>
<td>Cl.$NO_MATCH_WARNING&lt;br&gt;Displays a warning message on error output when a wildcard does not match existing pathnames.</td>
</tr>
<tr>
<td>CL_$NO_MATCH_WARNING</td>
<td>CL_$NO_MATCH_OK&lt;br&gt;Displays no warning or error message when a wildcard does not match existing pathnames.</td>
</tr>
<tr>
<td>Displays a warning message on error output when a wildcard does not match existing pathnames.</td>
<td>CL_$NO_MATCH_ERROR&lt;br&gt;Displays an error message on error output, and terminates the program when a wildcard does not match an existing pathname.</td>
</tr>
<tr>
<td>CL_$VERIFY_NONE CL_$GET_NAME does not verify any names with the user.</td>
<td>CL_$VERIFY_WILD&lt;br&gt;CL_$GET_NAME verifies only names resolved by wildcards with the user.</td>
</tr>
<tr>
<td>CL_$VERIFY_ALL CL_$GET_NAME verifies all names with the user.</td>
<td></td>
</tr>
<tr>
<td>Default Option</td>
<td>Mutually-Exclusive Alternatives</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>CL_ $STAR_ NAMES</td>
<td>CL_ $NO_ STAR_ NAMES</td>
</tr>
<tr>
<td>Allows user to specify names-files with the * operator.</td>
<td>Does not allow user to specify names-files. Treats the *** just like any other character.</td>
</tr>
<tr>
<td>CL_ $KEYWORD_ DELIM</td>
<td>CL_ $NO_ KEYWORD_ DELIM</td>
</tr>
<tr>
<td>Prevents CL_ $GET_ ARG and CL_ $GET_ NAME from returning an &quot;unused&quot; flag as an argument. The calls return FALSE if they find a flag.</td>
<td>Causes no special treatment of unread flags by CL_ $GET_ ARG and CL_ $GET_ NAME. The procedures return the flags as if they were arguments.</td>
</tr>
<tr>
<td>CL_ $DASH_ NOP</td>
<td>CL_ $DASH_ NAMES</td>
</tr>
<tr>
<td>The CL treats the hyphen <em>-</em> as a name. Normally, the hyphen is an identifier for the standard input stream.</td>
<td>The CL reads names from standard input when it finds a hyphen.*</td>
</tr>
<tr>
<td>CL_ $DASH_ DFT_ NOP</td>
<td>CL_ $NAME_ DFT_ STDIN</td>
</tr>
<tr>
<td>Suppresses any special action when there are no arguments on the command line.</td>
<td>Causes the CL to read names from standard input if no arguments appear on the command line.*</td>
</tr>
<tr>
<td>CL_ $NO_ COMMENTS</td>
<td>CL_ $COMMENTS</td>
</tr>
<tr>
<td>Causes no special treatment of characters enclosed in brackets.</td>
<td>Causes CL to ignore all characters enclosed in brackets when reading a names-file. Do not use this in commands that accept derived names because you must use brackets to specify the tag expressions you want to use in a derived name.</td>
</tr>
</tbody>
</table>

* Made available to remain compatible with previous software releases. Obsolete for new software development.

**program_name**

Name of program, in character string format. CL uses this name when reporting any errors.

**program_len**

Length of "program_name". This is a 2-byte integer.

**USAGE**

Use this call to initialize the CL. This call reads the command line and any names-files that the user supplies and builds an internal token list that contains these tokens. All subsequent CL calls refer to this token list, not the command line.
CL_$MATCH

Compares a token against a specified string and returns TRUE if they match.

FORMAT

match := CL_$MATCH(pattern, token, token_len)

RETURN VALUE

match

Returns TRUE if it finds a match; returns FALSE if not.

INPUT PARAMETERS

pattern

Pattern with which to compare the token. This is a lowercase UNIV character string in the form of "required[optional]."

"required" represents the characters of the flag that the user must type. "optional" represents those characters that are optional. If the user specifies any optional characters, they all must be specified.

If you do not specify any optional characters, you must supply an empty bracket, [].

For example, "q[uit]" matches "q" or "quit." "go[]" matches "go."

token

Character string to compare against the pattern. This is a lowercase character string in UNIV character string format.

token_len

The number of characters to place in "token." This is a 2-byte integer.

USAGE

Use this call to compare a string against a specification. For example, this allows you to perform an action as soon as the user types the command. The following example uses CL_$MATCH in an interactive parsing loop to exit the program as soon as the user types a response:

cl_$parse_line ( line, line_len );

{ Terminate program if user types a q[uit]. }

IF cl_$get_arg (cl_$first, token, token_len, sizeof(token_len) THEN

IF cl_$match ('q[uit]', token, token_len THEN

   pgm_$exit

ELSE IF cl_$match ( ...
CL$_$PARSE$_$ARGS

Takes the specified argument vector, and makes it the current vector for the CL to parse. The CL discards any previous arguments. All subsequent CL calls operate on this argument list.

FORMAT

CL$_$PARSE$_$ARGS(arg_count, arg_vector)

INPUT PARAMETERS

arg_count
Number of arguments in the argument vector. This is a 2-byte integer.

arg_vector
Address of the argument vector to parse, in CL$_$ARGV format. This is a 4-byte integer. It is a UNIV array of pointers to arguments in a record.

USAGE

Use this call when you use PGM$_$GET$_$ARGS to get arguments from the command line. Use CL$_$SETUP to initialize the CL when you want to supply the lines to parse.

Note that this call disregards the first argument on the command line because it assumes it is the command name.
CL_$PARSE_INPUT

Parses a line of text from a specified stream and passes the line to CL_$PARSE_LINE.

FORMAT

ok := CL_$PARSE_INPUT(stream_id, null_line)

RETURN VALUE

ok

Returns TRUE if it reads the line from the stream successfully; returns FALSE if it encounters an end-of-file. If any other errors occur, the CL prints an error message, and calls PGM_$EXIT to terminate the program.

INPUT PARAMETERS

stream_id
Stream ID of the stream from which the line is read, in STREAM_$ID_T format.

OUTPUT PARAMETERS

null_line
Indicates whether the line is NULL (that is, the line contains only the NEWLINE character). This is a Boolean value.

The call returns TRUE if the line read is NULL; returns FALSE if the line is not NULL.

USAGE

You can use this call and CL_$PARSE_LINE so that you can use other CL calls to read tokens that the user inputs interactively.

Use CL_$SETUP to initialize the CL when you want to supply the lines to parse.
CL_$PARSE_LINE

Parses a line of text and creates a new token list. All subsequent CL calls operate on this line.

FORMAT

CL_$PARSE_LINE(text, null_line)

INPUT PARAMETERS

text
Character string to be parsed. This is in UNIV character string format.

text_len
Number of characters in "text." This is a 2-byte integer.

USAGE

Use this call if the lines of text you want to parse are contained in files. Use CL_$SETUP to initialize the CL when you want to supply the lines to parse.
CL_$REREAD

Marks the entire token list "unused," so that you can reread the entire token list.

FORMAT
CL_$REREAD

USAGE
Use this call to reread the entire command line. If you want to reread only the flags on the
token list, use CL_$REREAD_FLAGS. If you want to reread only the names, use
CL_$REREAD_NAMES.
CL_$REREAD_FLAGS

Locates all the "used" flags on the token list and marks them "unused." This allows you to reread all the flags.

FORMAT
CL_$REREAD_FLAGS

USAGE

Use this call to reread the flags on token list after having read them using CL_$GET_FLAG, CL_$CHECK_FLAG or CL_$GET_ENUM_FLAG.

To reread the entire token list, use CL_$REREAD.
CL_$REREAD_NAMES

 Locates all the "used" names on the token list and marks them "unused." This allows you to reread all the names using CL_$GET_NAME.

FORMAT
CL_$REREAD_NAMES

USAGE

 Use this call to reread the names on the token list after having read them with CL_$GET_NAME.

 When you call CL_$GET_NAME after this call, it returns the first name on the token list.

 To reread the entire token list, use CL_$REREAD.
CL_$RESET_OPTIONS

Replaces the previously defined CL option set with the set specified in this call.

FORMAT

CL_$RESET_OPTIONS([cl_option])

INPUT PARAMETERS

cl_option

Set of options, in CL_$OPT_SET_T format, that tells the CL how to read the command line. This is a 4-byte integer. These options control how the CL interacts with the user.

You must specify all the options you want in effect, except the default options.

The chart below lists the default CL options and their alternatives. Note that you can take either one of the default options, or specify one of the corresponding alternatives.

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$WILDCARDS</td>
<td>CL_$NO_WILDCARDS</td>
</tr>
<tr>
<td>Causes CL_$GET_NAME to resolve</td>
<td>Causes CL_$GET_NAME to return wildcard-names</td>
</tr>
<tr>
<td>wildcards and return the resolved</td>
<td>verbatim; it does not resolve them.</td>
</tr>
<tr>
<td>names.</td>
<td></td>
</tr>
<tr>
<td>CL_$NO_MATCH_WARNING</td>
<td>CL_$NO_MATCH_OK</td>
</tr>
<tr>
<td>Displays a warning message on</td>
<td>Displays no warning or error message when a wildcard</td>
</tr>
<tr>
<td>error output when a wildcard</td>
<td>does not match existing pathnames.</td>
</tr>
<tr>
<td>does not match existing pathnames.</td>
<td></td>
</tr>
<tr>
<td>CL_$VERIFY_NONE</td>
<td>CL_$VERIFY_WILD</td>
</tr>
<tr>
<td>CL_$GET_NAME does not verify</td>
<td>CL_$VERIFY_ALL</td>
</tr>
<tr>
<td>any names with the user.</td>
<td>CL_$GET_NAME verifies all names with the user.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Apollo Confidential
## Default Option

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CL</strong>_$STAR$ NAMES **</td>
<td>Allows user to specify names-files with the &quot;&quot; operator.</td>
</tr>
<tr>
<td><strong>CL</strong>_$NO$ STAR$ NAMES **</td>
<td>Does not allow user to specify names-files. Treats the &quot;&quot; just like any other character.</td>
</tr>
<tr>
<td><strong>CL</strong>_$KEYWORD$ _DELIM **</td>
<td>Prevents <strong>CL</strong>_$GET$ <em>ARG and <strong>CL</strong></em>$GET$ _NAME from returning an &quot;unused&quot; flag as an argument. The calls return FALSE if they find a flag.</td>
</tr>
<tr>
<td><strong>CL</strong>_$NO$ KEYWORD$ _DELIM **</td>
<td>Causes no special treatment of unread flags by <strong>CL</strong>_$GET$ <em>ARG and <strong>CL</strong></em>$GET$ _NAME. The procedures return the flags as if they were arguments.</td>
</tr>
<tr>
<td><strong>CL</strong>_$DASH$ _NOP **</td>
<td>The <strong>CL</strong> treats the hyphen &quot;-&quot; as a name. Normally, the hyphen is an identifier for the standard input stream.</td>
</tr>
<tr>
<td><strong>CL</strong>_$DASH$ _DFT$ _NOP **</td>
<td>Suppresses any special action when there are no arguments on the command line.</td>
</tr>
<tr>
<td><strong>CL</strong>_$NO$ _COMMENTS **</td>
<td>Causes no special treatment of characters enclosed in brackets.</td>
</tr>
<tr>
<td><strong>CL</strong>_$COMMENTS **</td>
<td>Causes <strong>CL</strong> to ignore all characters enclosed in brackets when reading a names-file. Do not use this in commands that accept derived names because you must use brackets to specify the tag expressions you want to use in a derived name.</td>
</tr>
</tbody>
</table>

* Made available to remain compatible with previous software releases. Obsolete for new software development.

## USAGE

Use this call to replace the set of **CL** options defined with the calls **CL**_$INIT$, **CL**_$SETUP$, or **CL**_$SET$ _OPTIONS. **CL**_$RESET$ _OPTIONS replaces the currently defined **CL** option set with the set specified in this call. If you want to simply add to the existing set of options, use **CL**_$SET$ _OPTIONS.
CL_$SETUP

CL_$SETUP

Initializes the command line handler, but does not load anything to parse. Use this call instead of CL_$INIT when you want to use one of the CL parse routines.

You must use either this call or CL_$INIT before any other CL calls.

FORMAT

CL_$SETUP([cl_option], program_name, program_len)

INPUT PARAMETERS

cl_option

Set of options, in CL_$OPT_SET_T format, that tells the CL how to read the command line. This is a 4-byte integer. These options control how the CL interacts with the user.

You can specify a default set of options with empty brackets, []. You need to specify only the options that are not default. The chart below lists the default CL options and their mutually-exclusive alternatives.

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
</table>
| CL_$WILDCARDS             | CL_$NO_WILDCARDS
Causes CL_$GET_NAME to resolve wildcard names. |
|                           | Causes CL_$GET_NAME to return wildcard-names verbatim; it does not resolve them. |
| CL_$NO_MATCH_WARNING     | CL_$NO_MATCH_OK
Displays a warning message on error output when a wildcard does not match existing pathnames. |
|                           | Displays no warning or error message when a wildcard does not match existing pathnames. |
|                           | CL_$NO_MATCH_ERROR
Displays an error message on error output, and terminates the program when a wildcard does not match an existing pathname. |
| CL_$VERIFY_NONE           | CL_$VERIFY_WILD
CL_$GET_NAME does not verify any names with the user. |
| CL_$GET_NAME verifies only names resolved by wildcards with the user. |
| CL_$VERIFY_ALL            | CL_$VERIFY_WILD
CL_$GET_NAME verifies all names with the user. |
### Default Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$STAR_NAMES</td>
<td>Allows user to specify names-files with the &quot;*&quot; operator.</td>
</tr>
<tr>
<td>CL_$KEYWORD_DELIM</td>
<td>Prevents CL_$GET_ARG and CL_$GET_NAME from returning an &quot;unused&quot; flag as an argument. The calls return FALSE if they find a flag.</td>
</tr>
<tr>
<td>CL_$DASH_NOP</td>
<td>The CL treats the hyphen &quot;-&quot; as a name. Normally, the hyphen is an identifier for the standard input stream.</td>
</tr>
<tr>
<td>CL_$NO_COMMENTS</td>
<td>Causes no special treatment of characters enclosed in brackets.</td>
</tr>
</tbody>
</table>

### Mutually-Exclusive Alternatives

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$NO_STAR_NAMES</td>
<td>Does not allow user to specify names-files. Treats the &quot;*&quot; just like any other character.</td>
</tr>
<tr>
<td>CL_$NO_KEYWORD_DELIM</td>
<td>Causes no special treatment of unread flags by CL_$GET_ARG and CL_$GET_NAME. The procedures return the flags as if they were arguments.</td>
</tr>
<tr>
<td>CL_$DASH_NAMES</td>
<td>The CL reads names from standard input when it finds a hyphen.</td>
</tr>
<tr>
<td>CL_$NAME_DFT_STDIN</td>
<td>Causes the CL to read names from standard input if no arguments appear on the command line.</td>
</tr>
<tr>
<td>CL_$COMMENTS</td>
<td>Causes CL to ignore all characters enclosed in brackets when reading a names-file. Do not use in commands that accept derived names; you must use brackets to specify tag expressions.</td>
</tr>
</tbody>
</table>

* Made available to remain compatible with previous software releases. Obsolete for new software development.

---

**program\_name**

Name of program, in UNIV character string format. CL uses this name when reporting any errors.

**program\_len**

Length of "program\_name." This is a 2-byte integer.

---

### USAGE

Use this call to initialize the CL when you want to provide the arguments to parse, rather than having the CL read arguments from the command line. You provide the CL with the arguments to parse by making subsequent calls to CL parse routines, CL\_$PARSE\_ARGS, CL\_$PARSE\_INPUT, or CL\_$PARSE\_LINE. You could use CL\_$INIT to initialize the CL and then call a parsing routine, but this is a cleaner and faster method.
CL_$SET_DERIVED_COUNT

Tells the CL how many derived names follow each wildcard-name or pathname on the command line.

FORMAT
CL_$SET_DERIVED_COUNT(count)

INPUT PARAMETERS

count
Number of derived names that you expect to follow a wildcard-name or pathname. This is a 2-byte integer.

USAGE
Use this call when you expect derived names. Usually, you specify a "count" of one; but potentially, you can have several derived names from one source name.

This call is optional, but recommended.
CL_$SET_NAME_PREFIX

Defines a character string that the CL adds to the beginning of each name read from the token list.

FORMAT

CL_$SET_NAME_PREFIX(prefix, prefix_len)

INPUT PARAMETERS

prefix
Prefix to insert in front of the name read from the token list. This is a UNIV character string.

prefix_len
Number of characters in "prefix." This is a 2-byte integer.

USAGE

Use this call to add text before a name read by CL_$GET_NAME. For example, use this call to add "/sys/print" before a file to be queued. The CL adds this prefix to the beginning of each argument read by CL_$GET_NAME before it resolves wildcards.
**CL$_$SET$_$OPTIONS**

Adds specified options to the set of CL options defined in the CL$_$INIT or CL$_$SETUP call.

**FORMAT**

CL$_$SET$_$OPTIONS([cl_option])

**INPUT PARAMETERS**

cl_option

Set of options, in CL$_$OPT SET T format, that tells the CL how to read the command line. This is a 4-byte integer. These options control how the CL interacts with the user.

You must specify all the options you want, except the default options.

The chart below lists the default CL options and their mutually-exclusive alternatives.

<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL$_$WILDCARDS</td>
<td>CL$_$NO_WILDCARDS</td>
</tr>
<tr>
<td>Causes CL$_$GET_NAME to resolve wildcards and return the resolved names.</td>
<td>Causes CL$_$GET_NAME to return wildcard-names verbatim; it does not resolve them.</td>
</tr>
<tr>
<td>CL$_$NO_MATCH_WARNING</td>
<td>CL$_$NO_MATCH_OK</td>
</tr>
<tr>
<td>Displays a warning message on error output when a wildcard does not match existing pathnames.</td>
<td>Displays no warning or error message when a wildcard does not match existing pathnames.</td>
</tr>
<tr>
<td>CL$_$VERIFY_NONE</td>
<td>CL$_$VERIFY_WILD</td>
</tr>
<tr>
<td>CL$_$GET_NAME does not verify any names with the user.</td>
<td>CL$_$VERIFY_ALL</td>
</tr>
<tr>
<td>CL$_$VERIFY_WILD</td>
<td>CL$_$VERIFY_ALL</td>
</tr>
<tr>
<td>CL$_$GET_NAME verifies only names resolved by wildcards with the user.</td>
<td>CL$_$VERIFY_ALL verifies all names with the user.</td>
</tr>
</tbody>
</table>

Apollo Confidential
<table>
<thead>
<tr>
<th>Default Option</th>
<th>Mutually-Exclusive Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_$STAR_NAMES</td>
<td>CL_$NO_STAR_NAMES</td>
</tr>
<tr>
<td>Allows user to specify names-files with the *** operator.</td>
<td>Does not allow user to specify names-files. Treats the *** just like any other character.</td>
</tr>
<tr>
<td>CL_$KEYWORD_DELIM</td>
<td>CL_$NO_KEYWORD_DELIM</td>
</tr>
<tr>
<td>Prevents CL_$GET_ARG and CL_$GET_NAME from returning an &quot;unused&quot; flag as an argument. The calls return FALSE if they find a flag.</td>
<td>Causes no special treatment of unread flags by CL_$GET_ARG and CL_$GET_NAME. The procedures return the flags as if they were arguments.</td>
</tr>
<tr>
<td>CL_$DASH_NOP</td>
<td>CL_$DASH NAMES</td>
</tr>
<tr>
<td>The CL treats the hyphen &quot;-&quot; as a name. Normally, the hyphen is an identifier for the standard input stream.</td>
<td>The CL reads names from standard input when it finds a hyphen.*</td>
</tr>
<tr>
<td>CL_$DASH_DFT_NOP</td>
<td>CL_$NAME_DFT_STDIN</td>
</tr>
<tr>
<td>Suppresses any special action when there are no arguments on the command line.</td>
<td>Causes the CL to read names from standard input if no arguments appear on the command line.*</td>
</tr>
<tr>
<td>CL_$NO_COMMENTS</td>
<td>CL_$COMMENTS</td>
</tr>
<tr>
<td>Causes no special treatment of characters enclosed in brackets.</td>
<td>Causes CL to ignore all characters enclosed in brackets when reading a names-file. Do not use this in commands that accept derived names because you must use brackets to specify the tag expressions you want to use in a derived name.</td>
</tr>
</tbody>
</table>

* Made available to remain compatible with previous software releases. Obsolete for new software development.

**USAGE**

Use this call to add options to the set of CL options that you defined when you initialized the CL with a call to CL_$INIT or CL_$SETUP. If you want to remove any options from the set, use CL_$RESET_OPTIONS.
**FORMAT**

CL_SET_STREAMS (stream_in, error_in, stream_out, err_out)

**INPUT PARAMETERS**

*stream_in*

Name of stream used for standard input, in STREAM_$ID_T format. The default value is stream_$stdin (0).

*error_in*

Name of stream used for error input, in STREAM_$ID_T format. The default value is stream_$errin (2).

*stream_out*

Name of stream used for standard output, in STREAM_$ID_T format. The default value is stream_$stdout (1).

*error_out*

Name of stream used for error output, in STREAM_$ID_T format. The default value is stream_$errout (3).

**USAGE**

Use this call to redirect the default input and output channels to the specified streams. This allows the user to specify another file for input or output.
CL_$SET_VERB

Defines a verb that the CL displays before each query message.

FORMAT

CL_$VERB(verb, verb_len)

INPUT PARAMETERS

verb

Text that the CL prints before each pathname in user query messages. This is a UNIV character string.

verb_len

Number of characters in "verb." This is a 2-byte integer.

USAGE

Use this call to add text before a pathname read by CL_$GET_NAME when you are expecting the CL to query users. When the CL queries users, it will write the verb, the pathname, and then a question mark.

The CL verifies all pathnames when you specify the CL_$VERIFY_ALL CL option, or when you use the CL_$VERIFY system call. It verifies wildcard matches if you specify CL_$VERIFY_WILD.
$SET_WILD_OPTIONS

Defines the wildcard options that tell the wildcard manager how to expand wildcards.

**FORMAT**

$SET_WILD_OPTIONS([wild_option])

**INPUT PARAMETERS**

wild_option

Set of options, in CL_$WILD_SET_T format, that tells the CL wildcard manager how to expand wildcard options. This is a 2-byte integer.

If you want to change the default set of wildcard options that the CL established at initialization, you must specify the all the options you want to set.

The CL wildcard options are:

- **CL_$WILD_FILES**
  Matches names of files (default).

- **CL_$WILD_DIRS**
  Matches names of directories (default).

- **CL_$WILD_LINKS**
  Matches names of links (default).

- **CL_$WILD_EXCLUSIVE**
  Matches the highest directory of a given wildcard. This is useful for commands that operate on entire directories, such as COPY_TREE, or WRITE_BACKUP. Since they operate on all subdirectories, there's no need to match further.

- **CL_$WILDCHASE_LINKS**
  Chases links that point to directories or files.

- **CL_$WILD FIRST**
  Stops at the first match of a given wildcard; otherwise, subsequent calls to CL_$GET_NAME expand all names that match the wildcard.

**USAGE**

Use this call to change the default set of CL wildcard options.
CL_$VERIFY

Prompts the user for a yes/no response. This call adds a question mark after the specified prompt.

FORMAT

answer := CL_$VERIFY(name, name_len)

RETURN VALUE

answer

Returns the user's response to the yes/no query in CL_$ANSWER_T format. This is a 2-byte integer.

The user's response can be either of the following values:

CL_$YES Include this pathname on the token list.
CL_$NO Ignore this pathname.
CL_$QUIT Stop prompting user to verify pathnames.
CL_$GO Include all the pathnames that you find on the token list without further query.

INPUT PARAMETERS

name

Name with which the user is prompted. This is a UNIV character string.

ame_len

Number of characters in "name." This is a 2-byte integer.

USAGE

Use this call to prompt the user for a yes/no response. It writes the name followed by a question mark. You can call CL_$SET_VERB before this call to include a text string with the prompt.
CL ERRORS

ERRORS

CL_$ARG_TOO_LONG
User supplied an argument that is longer than allowed. You specify the maximum length required at the time of the call.

CL_$DUPLICATE_SET_ELEMENT
User supplied a duplicate character, detected by the CL_$GET_SET call.

CL_$INVALID_DECIMAL_NUMBER
User supplied an invalid decimal number to be converted by the CL_$GET_NUM call.

CL_$INVALID_SET_ELEMENT
User supplied an invalid character, detected by the CL_$GET_SET call.

CL_$MISSING_REQ_DERIVED_NAME
User did not supply the required derived name after a keyword for the CL_$GET_FLAGGED_DERIVED_NAME call.

CL_$NO_MATCH_FOR_WILDCARD
There are no pathnames in the user's working directory that match the wildcard specified.

CL_$NOT_ENOUGH_ARGUMENTS
User did not supply enough arguments, detected by the CL_$CHECK_FLAG call.

CL_$TOO_MANY_ARGUMENTS
User supplied more arguments than the program can handle, detected by the CL_$CHECK_FLAG call.

CL_$UNPARSED_KEYWORD
User supplied a keyword that cannot be handled by program, detected by the CL_$CHECK_UNCLAIMED call.
**FU DATA TYPES**

**CONSTANTS**

**NAME_$$PNAMLEN_MAX** 256 Maximum length of a pathname.

**DATA TYPES**

**NAME_$$SPNAME_T**

An array of up to **NAME_$$PNAMLEN_MAX** (256) characters.

**FU_$$CONTEXT_T**

A 2-byte integer. Location of error returned by some FU calls. One of the following pre-defined values:

- **FU_$$SRC**
  Error occurred in the source object or tree.

- **FU_$$DST**
  Error occurred in the target object or tree.

- **FU_$$UNK**
  Unknown whether the error occurred in the source or target object or tree.

**FU_$$OPT_T**

A 2-byte integer. FU options passed to many FU calls. One of the following pre-defined values:

- **FU_$$LIST_FILES**
  List files operated on.

- **FU_$$LIST_DIRS**
  List directories operated on.

- **FU_$$LIST_LINKS**
  List links operated on.

- **FU_$$REPLACE**
  Replace target with a copy of the source.

- **FU_$$MERGE**
  Merge source and target if both are directories. For files and links with the same name in source and target, it deletes the target and replaces it with a copy of the source.

- **FU_$$COE**
  Continue to the next object if an error occurs while processing an object.

- **FU_$$PRINT_ERRORS**
  Print errors on the error output stream.
FU DATA TYPES

FU_$LIST_DEL
List objects deleted as a result of a replace operation. Obsolete; use FU_$LIST_D_DIRS,
FU_$LIST_D_FILES,
FU_$LIST_D_LINKS instead.

FU_SHHELP
Display detailed usage information. It has no meaning under AEGIS, used for a Boot Shell
utility only.

FU_$QUIT
Has no meaning under AEGIS, used for a Boot Shell utility only.

FU_$BEF_TIME
For copy tree, copy only those objects whose dtm (date/time last modified) is before the
given date and time.

FU_$AFT_TIME
For copy tree, copy only those objects whose dtm (date/time last modified) is after the
given date and time.

FU_$FORCE_DEL
Force deletion of a target during a replace operation if user has owner "P" rights.

FU_$FORCE
Force a copy in FU_$MOVE if the source and target files are not located on the same
volume.

FU_$DAACL
Assign default ACL to target files. Target gets the default ACL of the parent (destination)
directory.

FU_$SAACL
Assign ACL of source object to target object. Target gets the same ACL as the source
object.

FU_$RENAME
Change the name of existing object with the target pathname before creating copy. If
target name is in use and cannot be deleted during a replace operation, it appends today's
date to the target pathname.

FU_$LIST_D_FILES
List deleted files.
FU DATA TYPES

FU_$LIST_D_DIRS
List deleted directories.

FU_$LIST_D_LINKS
List deleted links.

FU_$MERGE_DST
Merge source and target if both are directories. For files and links with the same name in source and target, the target remains unchanged.

FU_$USE_PRESERVE
Reserved.

FU_$SUBS
Retain the source ACL for objects which belong to protected subsystems.

FU_$PRESERVE_DT
Preserve the source dtm (date/time last modified) and dtu (date/time last used).

FU_$SPARSE
Reserved.

FU_$DEL_WHEN_UNLKD
Delete object when it becomes unlocked.

FU_$OPT_SET_T
A 2-byte integer. A set of FU options in FU_$OPT_T format. For a list of options, see FU_$OPT_T above.

STATUS_$T
A status code. The diagram below illustrates the STATUS_$T data type:

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>0</td>
<td>31</td>
<td>integer</td>
</tr>
<tr>
<td>fail</td>
<td>24</td>
<td>31</td>
<td>integer</td>
</tr>
<tr>
<td>subsys</td>
<td>16</td>
<td>24</td>
<td>integer</td>
</tr>
<tr>
<td>mdoc</td>
<td>0</td>
<td>16</td>
<td>integer</td>
</tr>
<tr>
<td>code</td>
<td>0</td>
<td>0</td>
<td>all</td>
</tr>
</tbody>
</table>
```
Field Description:

ALL
All 32 bits in the status code.

FAIL
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

SUBSYS
The subsystem that encountered the error (bits 24 - 30).

MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).

### TIME_$CLOCKH_T

Internal representation of time. The high 32 bits of the TIME_$CLOCKH_T data type. The diagram below illustrates the TIME_$CLOCKH_T data type:

<table>
<thead>
<tr>
<th>Total</th>
<th>byte:</th>
<th>offset</th>
<th>field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size: 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predefined Record:

time_$clockh_t

<table>
<thead>
<tr>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
</tr>
</tbody>
</table>

Low 16 bits of the clock.

Field Description:

HIGH
High 32 bits of the clock.

LOW
Low 16 bits of the clock.
FU\_\$CMP\_TREE

Compare a source tree to a target tree.

FORMAT

FU\_\$CMP\_TREE(source\_pathname, source\_name\_len, target\_pathname, target\_name\_len, fu\_options, error\_pathname, error\_name\_len, status)

INPUT PARAMETERS

source\_pathname
Pathname of the source file to be compared, in NAME\_$PNAME\_T format. This is an array of up to 256 characters.

The source file must be the same type of object as the target.

source\_name\_len
Number of characters in "source\_pathname." This is a 2-byte integer.

target\_pathname
Pathname of the target file to be compared, in NAME\_$PNAME\_T format. This is an array of up to 256 characters.

The target file must be the same type of object as the source.

target\_name\_len
Number of characters in "target\_pathname." This is a 2-byte integer.

fu\_options
FU compare options in FU\_\$OPT\_SET\_T format. This is a 2-byte integer. Specify any combination of the following predefined values:

FU\_\$COE Continues to the next file, if an error occurs while processing a file.

FU\_\$LIST\_DIRS Lists directories as they are compared.

FU\_\$LIST\_FILES Lists files as they are compared.

FU\_\$LIST\_LINKS Lists links as they are compared.

FU\_\$PRINT\_ERRORS Displays errors to the error output stream.
INPUT/OUTPUT PARAMETERS

error_pathname
Pathname of the file where an error occurred, if any, in NAME_ $PNAME_ T format.
This is an array of up to 256 characters.
This is valid only if "status" does not equal zero.

error_pathname_len
Length of "error_pathname." This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_ $T format. This data type is 4 bytes long. See the FU
Data Types section for more information.

USAGE

Use this call to compare all the objects of a source tree against all objects in a target tree.
This call reports any objects catalogued in the source that do not also appear in the target.
However, it does not compare the target to the source, so it does not list objects that appear
in the target that do not appear in the source.

FU_ $CMP_ TREE compares objects byte-by-byte. If it encounters a difference, it reports
the difference, stops comparing that file and goes on to compare the next objects in the tree.
Copies a file from the source pathname to the target pathname.

**FORMAT**

```
FU_COPY_FILE(source_pathname, source_name_len, target_pathname, target_name_len, fu_options, error_context, status)
```

**INPUT PARAMETERS**

- **source_pathname**
  Pathname of the source file to be copied, in NAME_$PNAME_T format. This is an array of up to 256 characters.
  If the source name is a link name, it resolves the link, and copies the file to which the link refers.

- **source_name_len**
  Number of characters in "source_pathname." This is a 2-byte integer.

- **target_pathname**
  Pathname of the target file, in NAME_$PNAME_T format. This is an array of up to 256 characters.
  If pathname is a directory, it copies the source to the target directory. You cannot specify a link as a target name.

- **target_name_len**
  Number of characters in "target_pathname." This is a 2-byte integer.

- **fu_options**
  FU copy options in FU_$OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following predefined values:
  - **FU_SCOE** continues to the next file, if an error occurs while processing a file.
  - **FU_SDAACL** assigns the target file's ACL. The target file gets the default ACL of the parent (destination) directory. Invalid if **FU_SACACL** is set.
  - **FU_SDEL_WHEN_UNLKD** deletes object when it becomes unlocked as a result of a replace operation. (That is, if the user set the **FU_SREPLACE** option.)
  - **FU_SFORCE_DEL** forces deletion of a target during a replace operation if user has protect ("P") rights. (That is, if the user set the **FU_SREPLACE** option.)
  - **FU_SLIST_D_FILES** lists files deleted as a result of a replace operation. (That is, if the user set the **FU_SREPLACE** option.)
  - **FU_SLIST_FILES** lists files copied.
FU\_$PRESERVE\_DT
Preserves the source file's dtm (date/time last modified) and dtu (date/time last used) if the user set the FU\_$REPLACE option and the file was copied.

FU\_$PRINT\_ERRORS
Displays errors to the error output stream.

FU\_$RENAME Changes the name of existing object with the target pathname before making a copy. If target name is in use and cannot be deleted during a replace operation, it appends today's date to the target pathname.

FU\_$REPLACE Replaces the target with a copy of the source.

FU\_$SACL Assigns the target file's ACL. The target file gets the same ACL as the source file. Invalid if FU\_$DACL is set.

FU\_$SUBS Retains the source ACL for objects which belong to protected subsystems.

OUTPUT PARAMETERS

error\_context
Indicates where an error occurred, in FU\_$CONTEXT\_T format. This is a 2-byte integer. On error, the call can return any one of the following predefined values:

FU\_$SRC Error occurred in the source object.

FU\_$DST Error occurred in the target object.

FU\_$UNK Error undefined.

This is valid only if "status" does not equal zero.

status
Completion status, in STATUS\_T format. This data type is 4 bytes long. See the FU Data Types for more information.

USAGE
Use this call to copy a source file to a target file. This call copies only files; use FU\_$COPY\_TREE to copy directories and their subordinate objects.
FU_$COPY_TREE

Copies, merges, and replaces files, directories, and links.

FORMAT

FU_$COPY_TREE(source_pathname, source_name_len, target_pathname,
            target_name_len, fu_options, before_time, after_time,
            error_pathname, error_pathname_len, status)

INPUT PARAMETERS

source_pathname
Pathname of the source file, link, or directory tree to be copied, in NAME_$PNAME_T format. This is an array of up to 256 characters.

source_name_len
Number of characters in "source_pathname." This is a 2-byte integer.

target_pathname
Pathname of the target file to be created, replaced, or merged, in NAME_$PNAME_T format. This is an array of up to 256 characters.

The target pathname can be derived from the source pathname. It cannot be a link, logical volume entry directory, or network root directory.

target_name_len
Number of characters in "target_pathname." This is a 2-byte integer.

fu_options
FU options, in FU_$OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following predefined values:

FU_$AF_TIME Copies only those objects whose dtm (date/time last modified) is after the given date and time, in TIME_$CLOCKH_T format.

FU_$BF_TIME Copies only those objects whose dtm (date/time last modified) is before the given date and time, in TIME_$CLOCKH_T format.

FU$_COE Continues to the next file, if an error occurs while processing a file.

FU$_DACL Assigns the target directory's ACL. Each directory has its own ACL plus two default ACLs, one for its files and another for its subdirectories. Each subdirectory and file gets the target directory's default ACLs.

FU$_DEL_WHEN_UNLKD Deletes the object when it becomes unlocked as a result of a replace operation. (That is, if the user set the FU$_REPLACE option.)

FU$_FORCE_DEL Forces deletion of a target during a replace operation if user has protect (*P*) rights. (That is, if the user set the FU$_REPLACE option.)

FU$_LIST_D_FILES Lists files deleted as a result of a replace operation.
FU_$LIST_D_DIRS
Lists directories deleted as a result of a replace operation.

FU_$LIST_D_LINKS
Lists links deleted as a result of a replace operation.

FU_$LIST_DIRS
Lists directories as they are copied.

FU_$LIST_FILES
Lists files as they are copied.

FU_$LIST_LINKS
Lists links as they are copied.

FU_$MERGE
Merges the source and target if both are directories. If the target exists, it merges the source into the target, replacing files and links, and combining directories. If the target does not exist, FU_$COPY_TREE duplicates the source as the target.

If both source and target are directories, FU_$COPY_TREE compares their contents, object by object. Objects that exist in the source but not in the target are created in the target. Objects that exist in the target but not in the source remain unchanged.

If files and links have the same name in the source and target, FU_$COPY_TREE deletes the target and replaces it with a copy of the source. If directories have the same name in both source and target, it merges them.

If the source and target are not both directories, FU_$COPY_TREE deletes the target and replaces it with the source.

FU_$MERGE_DST
Merges source and target if both are directories. It works the same as FU_$MERGE except that files and links with the same name in both the source and target remain unchanged in the target.

FU_$PREPARE_DT
Preserves the source file's dtm (date/time last modified) and dtu (date/time last used) if the user set the FU_$REPLACE option and the file was copied.

FU_$PRINT_ERRORS
Displays errors to the error output stream.

FU_$RENAME
Changes the name of existing object with the target pathname before making a copy. If target name is in use and cannot be deleted during a replace operation, it appends today's date to the target pathname.

FU_$REPLACE
Replaces the target with the source. It deletes the tree starting at the target pathname and copies the entire source tree in its place. If the target pathname does not exist, it creates one and duplicates the source.
FU$_$COPY$_$TREE

FU$_$SACL Assigns the target directory's ACL. Each subdirectory and file gets the same ACL as the source directory.

FU$_$$SUBS Retains the source ACL for objects which belong to protected subsystems.

before$_$time
Specified time, in TIME$_$$CLOCKH$_$T format. This data type is 4 bytes long.
Copies only those files whose dtm (date/time last modified) is before the given date and time.

after$_$time
Specified time, in TIME$_$$CLOCKH$_$T format. This data type is 4 bytes long.
Copies only those files whose dtm (date/time last modified) is after the given date and time.

OUTPUT PARAMETERS

error$_$pathname
Returns the pathname of the file where an error occurred, in NAME$_$$PNAME$_$T format. This is an array of up to 256 characters.
This is valid only if "status" does not equal zero.

error$_$pathname$_$len
Length of "error$_$pathname." This is a 2-byte integer.

status
Completion status, in STATUS$_$T format. This data type is 4 bytes long. See the FU Data Types section for more information.

USAGE
Use this call to copy directories, subordinate objects, and links to a target directory. This is most useful in calls where you want to copy any pathname, and do not care whether the pathname is a file or tree.
If you want to copy files only, use FU$_$COPY$_$FILE.
**FU\_DELETE\_FILE**

Deletes a specified file.

**FORMAT**

`FU\_DELETE\_FILE(pathname, pathname\_len, force, delete\_when\_unlocked, status)`

**INPUT PARAMETERS**

*pathname*
  Pathname of the source file to be deleted, in NAME\_\$PNAME\_T format. This is an array of up to 256 characters.

*pathname\_len*
  Number of characters in "pathname." This is a 2-byte integer.

*force*
  If TRUE, forces file deletion if user has owner rights, even if user does not have delete rights. This is a Boolean value.

*delete\_when\_unlocked*
  If TRUE, it deletes delete the file when it becomes unlocked. This is a Boolean value.

**OUTPUT PARAMETERS**

*status*
  Completion status, in STATUS\_\$T format. This data type is 4 bytes long. See the FU Data Types section for more information.

**USAGE**

Use this call to delete a file. It is similar to NAME\_\$DELETE\_FILE except that it allows you to delete locked objects.
FU_$DELETE_TREE

Deletes a directory and all its descendants.

FORMAT

FU_$DELETE_TREE(pathname, pathname_len, fu_options, error_pathname, error_pathname_len, status)

INPUT PARAMETERS

pathname
Pathname of the directory to be deleted, in NAME_$PNAME_T format. This is an array of up to 256 characters.

If *pathname* is a directory, it deletes the directory and all subordinate objects (subdirectories, links, and files). If *pathname* is a link, it deletes the link, but has no effect on the files and directories named by the link.

pathname_len
Number of characters in *pathname.* This is a 2-byte integer.

fu_options
FU delete options in FU_$OPT_SET_T format. This is a 2-byte integer. Specify any combination of the following predefined values:

FU_$COE
Continues to the next file, if an error occurs while processing a file.

FU_$DEL_WHEN_UNLKD
Deletes object when it becomes unlocked.

FU_$FORCE_DEL
Forces deletion if user has protect (*P*) rights.

FU_$LIST_D_FILES
Lists files deleted.

FU_$LIST_D_DIRS
Lists directories deleted.

FU_$LIST_D_LINKS
Lists links deleted.

FU_$LIST_DIRS
Lists directories as they are copied.

FU_$LIST_DIRS
Lists directories as they are deleted.

FU_$LIST_FILES
Lists files as they are deleted.

FU_$LIST_LINKS
Lists links as they are deleted.
FU_$PRINT_ERRORS
Displays errors to the error output stream.

OUTPUT PARAMETERS

error_pathname
Returns the pathname of the file where an error occurred, in NAME_$PNAME_T format.
This is an array of up to 256 characters.

This is valid only if "status" does not equal zero.

error_pathname_len
Length of "error_pathname." This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the FU Data Types section for more information.

USAGE

Use this call to delete a directory, and all the files, links and subdirectories within it. This call is most useful when you want to delete a pathname, and you do not care if the pathname is a file or directory.
FU_$INIT

FU_$INIT

Initializes the file and tree utility (FU). You must use this call before any other FU calls.

FORMAT
FU_$INIT

USAGE

Use this call to initialize the FU. This call allocates read/write storage space for subsequent FU calls. You can release the storage with FU_$RELEASE_STORAGE before terminating your program.
FU_$MOVE_FILE

Moves a file to a different location in the naming tree.

FORMAT

FU_$MOVE_FILE(source_pathname, source_name_len, target_pathname, 
   target_name_len, fu_options, error_context, status)

INPUT PARAMETERS

source_pathname
   Pathname of the source file to be moved, in NAME_$PNAME_T format. This is an 
   array of up to 256 characters.

source_name_len
   Number of characters in "source_pathname." This is a 2-byte integer.

target_pathname
   Pathname of the new file location, in NAME_$PNAME_T format. This is an array of up 
   to 256 characters.

target_name_len
   Number of characters in "target_pathname." This is a 2-byte integer.

fu_options
   FU options in FU_$OPT_SET_T format. This is a 2-byte integer. Specify any 
   combination of the following predefined values:
   
   FU_$COE Continues to the next file, if an error occurs while processing a file.

   FU_$DACL Assigns the target file's ACL. The target file gets the default ACL of its 
   parent (destination) directory. Invalid if FU_$SACL is set.

   FU_$DEL_WHEN_UNLKD Deletes the object when it becomes unlocked as a result of a replace 
   operation. (That is, if user set the FU_$REPLACE option.).

   FU_$FORCE Forces a copy of the target if the source and target are not located on the 
   same volume.

   FU_$LIST_D_FILES Lists files deleted as a result of a replace operation.

   FU_$LIST_D_DIRS Lists directories deleted as a result of a replace operation.

   FU_$LIST_D_LINKS Lists links deleted as a result of a replace operation.

   FU_$LIST_DIRS Lists directories moved.

   FU_$LIST_FILES Lists files moved.
FU_\$MOVE_FILE

FU_\$LIST_LINKS
Lists links moved.

FU_\$PREserve_DT
Preserves the source file's dtm (date/time last modified) and dtu (date/time last used) if the user set the FU_\$REPLACE option and the file was copied.

FU_\$PRINT_ERRORS
Displays errors to the error output stream.

FU_\$RENAME Changes the name of existing object with the target pathname before making a copy. If the target object is in use and cannot be deleted during a replace operation, it appends today's date to the target pathname.

FU_\$REPLACE Replaces the target with a copy of the source.

FU_\$SACL Assigns the target file's ACL. The target file gets the same ACL as the source file. Invalid if FU_\$DACL is set.

FU_\$SUBS Retains the source ACL for objects that belong to protected subsystems.

OUTPUT PARAMETERS

error_context
Indicates where an error occurred, in FU_\$CONTEXT_T format. This is a 2-byte integer. On error, the call can return any one of the following predefined values:

FU_\$SRC Error occurred in the source object.

FU_\$DST Error occurred in the target object.

FU_\$UNK Error undefined.

This is valid only if "status" does not equal zero.

status
Completion status, in STATUS_\$T format. This data type is 4 bytes long. See the FU Data Types section for more information.

USAGE

Use this call to move a specified file to another location in the naming tree. You can also use FU_\$MOVE_FILE to move a directory name, if the directory is located on the same volume.
FU_$RELEASE_STORAGE

Releases read/write storage used by FU calls.

FORMAT

FU_$RELEASE_STORAGE

USAGE

Use this call to release the read/write storage used by FU calls. FU allocates storage during copy operations, so you would usually use this call after your program performs numerous copy operations.

You can use this call any time after you initialize the FU with a FU_$INIT system call.
**FU\_$RENAME\_UNIQUE**

**FU\_$RENAME\_UNIQUE**

Renames a pathname to create a unique name by appending today's date to the pathname.

**FORMAT**

**FU\_$RENAME\_UNIQUE**(*pathname*, *pathname\_len*, *fu\_options*, *status*)

**INPUT PARAMETERS**

**pathname**
Pathname to be changed, in NAME\_$PNAME\_T format. This is an array of up to 256 characters.

**pathname\_len**
Number of characters in "pathname." This is a 2-byte integer.

**fu\_options**
FU options in FU\_$OPT\_SET\_T format. This is a 2-byte integer. Specify any combination of the following predefined values:

- **FU\_$COE**         Continues to the next file, if an error occurs while processing a file.
- **FU\_$LIST\_DIRS**  Lists directories operated on.
- **FU\_$LIST\_FILES** Lists files operated on.
- **FU\_$LIST\_LINKS** Lists links operated on.
- **FU\_$PRINT\_ERRORS** Displays errors to the error output stream.

**OUTPUT PARAMETERS**

**status**
Completion status, in STATUS\_$T format. This data type is 4 bytes long. See the FU Data Types section for more information.

**USAGE**

Use this call to rename a file so it will have a unique pathname. The call appends a period and today's date and time to the specified pathname.
**FU\_SET\_PROG\_NAME**

Identifies the program name when the FU manager reports any errors.

**FORMAT**

FU\_SET\_PROG\_NAME(pathname, pathname\_len)

**INPUT PARAMETERS**

pathname
Name of program, in NAME\_\$PNAME\_T format. This is an array of up to 256 characters.

pathname\_len
Number of characters in "pathname." This is a 2-byte integer.

**USAGE**

Use this call to identify your program. The FU manager returns this program name when reporting any errors that occur.
FU ERRORS

ERRORS

FU_$SCANT_PROCESS
Cannot delete or copy system directory.

FU_$COMPARE_FAILED
Compare failed, detected by FU_$CMP_TREE.

FU_$DEST_IN_SOURCE
Target file of FU_$COPY_TREE contained in source.

FU_$DIFF_VOLS
Cannot move objects across volumes, detected by FU_$MOVE_FILE.

FU_$NOT_LEAF
File is a directory, detected by FU_$COPY_FILE.

FU_$SAME_OBJECT
Cannot copy source over itself.

FU_$UNREC_NSTYPE
Naming server entry type is unknown.

FU_$UNREC_SYSTYPE
Type of system object is unknown, detected by FU_$CMP_TREE.
CONSTANTS

LOADER_TABLE_SIZE 2048 Maximum number of sections allowed in an array of the data type, PM_$SECT_INFO.

NAME_$PNAMLEN_MAX 256 Maximum length of a pathname.

DATA TYPES

NAME_$PNAME_T An array of up to NAME_$PNAMLEN_MAX (256) characters.

PM_$OPTS A 2-byte integer. Set of options that define how an object module gets loaded with PM_$LOAD. One of the following pre-defined values:

- PM_$COPY_PROC Causes PM_$LOAD to copy the object module into read/write storage so that you can write to the object module without changing the original.

- PM_$INSTALL Tells PM_$LOAD that it is loading a library containing global variables, not a program. PM_$LOAD makes all the global entry points that were marked at binding available to other programs.

- PM_$NO_UNRESOLVEDS Causes PM_$LOAD to report an error if there are any unresolved global variables.

- PM_$LOAD_GLOBALS Reserved.

- PM_$INSTALL_SECTION Tells PM_$LOAD that it's loading a library containing global sections. PM_$LOAD makes these global sections, which were marked at binding, available to all programs.

- PM_$LOAD_WRITABLE Causes PM_$LOAD to load the object module with read/write access. Normally, the object module has read/only access. Specify this option when you want to write to the object module without copying it first.

PM_$LOAD_OPTS A 4-byte integer. A set of LOADER options in PM_$OPTS format. For a list of options, see PM_$OPTS above.
LOADER DATA TYPES

PM_$SECT_INFO

A record of information within the PM_$LOAD_INFO data type. The diagram below illustrates the PM_$SECT_INFO data type:

Total Size: 40

field name

<table>
<thead>
<tr>
<th>byte offset</th>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>char</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>32:</td>
<td>integer</td>
<td>loc</td>
<td></td>
</tr>
<tr>
<td>36:</td>
<td>integer</td>
<td>len</td>
<td></td>
</tr>
</tbody>
</table>

NAME
The name of the section, a character array of up to 32 elements.

LOC
A 4-byte integer. Location of section information.

LEN
A 4-byte integer. Length of section.

PM_$LOAD_INFO

An argument returned by PM_$LOAD. The diagram below illustrates the PM_$LOAD_INFO data type:

Total Size: 81926

field name

<table>
<thead>
<tr>
<th>byte offset</th>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>integer</td>
<td>start_addr</td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>integer</td>
<td>n_sects</td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>char</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>81886: (6 + (2048 x 40) - 40)</td>
<td>sects:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Apollo Confidential
Field Description:

START_ADDRESS
A UNIV_PTR indicating the start address of the object module; i.e., the first instruction to execute. If the object is a library, the first instruction is the initialization point; if the object is a program, it is the main entry point.

N_SECTS
Number of sections contained in object module.

SECTS
Location of sections. An array of PM_$SECT_INFO records, up to LOADER_TABLE_SIZE (2048 bytes).

PM_$LOAD_INFO_PTR_T
A 4-byte integer. The address of a returned PM_$LOAD_INFO record.

STATUS_$T
A status code. The diagram below illustrates the STATUS_$T data type:

Field Description:

ALL
All 32 bits in the status code.

FAIL
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

SUBSYS
The subsystem that encountered the error (bits 24 - 30).
MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).

UNIV_PTR
A 4-byte integer. A pointer to allocated storage.
PM_$CALL

PM_$CALL
Invokes a program at the start address returned by PM_$LOAD.

FORMAT
long_int := PM_$CALL( start_address )

RETURN VALUE
long_int
Returns a 32-bit integer. This value has no meaning unless you expect a return value from
the invoked program. For details, see the chapter on process manager system calls in Part I
of this manual.

INPUT PARAMETERS
start_address
Starting address of the object module. This is a UNIV_PTR, which is the first field of the
PM_$LOAD_INFO record, returned by PM_$LOAD.

USAGE
This call invokes a program at the current program level, at the start address returned by
PM_$LOAD. The start address is the first field of the PM_$LOAD_INFO data type.
For example, if you declare *sec_info* of the type PM_$LOAD_INFO, you would
invoke the object module with the following statement:

    pm_junk := pm_call( sec_info.start_addr );

Before using this call, You must load the program with the PM_$LOAD system call.
Using these two system calls is similar to using PGM_$INVOKE, except that
PGM_$INVOKE creates a new program level and performs cleanup handling.
PM_\$LOAD

Loads a specified object module or library at the current program level.

FORMAT
PM_\$LOAD(program, program_len, loader_options, num_sects, load_info, status )

INPUT PARAMETERS

program
Name of the object to load, in NAME_\$PNAME_ T format. This is an array of up to 256 characters.

program_len
Number of characters in "name." This is a 2-byte integer.

loader_options
Loader options in PM_\$LOADER_ OPTS format. This is a 2-byte integer. Specify any combination of the following predefined values:

PM_\$COPY_PROC
Causes PM_\$LOAD to copy the object module procedure text to avoid changing the original object module. If you do not specify this option, PM_\$LOAD maps the object module directly.

PM_\$INSTALL Indicates that PM_\$LOAD is loading a library containing global variables, not a program. The entries are marked at binding.

PM_\$INSTALL_SECTIONS
Causes PM_\$LOAD to define global sections so that you can share information among programs.

PM_\$LOAD_WRITABLE
Causes PM_\$LOAD to load the object module as writable, allowing you read/write access. Normally, the object module has read/only access.

PM_\$NO_UNRESOLVEDS
Causes PM_\$LOAD to report an error if there are any unresolved global variables.

num_sects
Number of sections. This is a 2-byte integer. If "num_sects" is greater than zero, it is the maximum number of sections that you want information on, which is returned in "load_info." If "num_sects" is zero, PM_\$LOAD returns only the start address of the object module.

OUTPUT PARAMETERS

load_info
Returns the start address of the object module in PM_\$LOAD_INFO format. If "num_sects" does not equal zero, it also returns the name and location of each section in the object module. For more information on this data type, see the LOADER Data Types section.

Apollo Confidential

LOADER
PM_$LOAD

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOADER Data Types section for more information.

USAGE
Loads a library or program object module at the current program level. It converts the object module on the disk to executable form and returns the starting address of the module. You must invoke the program with the PM_$CALL system call.

Using these two system calls is similar to using PGM_$INVOKE, except that PGM_$INVOKE creates a new program level and performs cleanup handling.
ERRORS

KG_$NO_SPACE
   Not enough storage space for global entry points exported by this module for a
   read/write section.

LOADER_$BKPTS_IN_OBJ
   Leftover breakpoints exist in object module. This occurs when you set breakpoints in
   the original object module by using PM_$WRITABLE rather than making a copy
   with PM_$COPY_PROC. The breakpoints remain in the object module if it
   terminates abnormally.

LOADER_$DNx60_REQUIRED
   Attempted to execute an object that contains instructions specifically for the Dnx60
   series on a non Dnx60 machine.

LOADER_$M020_REQUIRED
   Attempted to execute an object that contains instructions specifically for the M020
   series on a non M020 machine.

LOADER_$M881_REQUIRED
   Attempted to execute an object that contains instructions specifically for the M881
   series on a non M881 machine.

LOADER_$NO_COPY_SPACE
   Not enough read/write storage space to copy the object module, detected because the
   PM_$COPY_PROC option was set.

LOADER_$NO_PROC_SPACE
   Not enough address space to map the object module.

LOADER_$NO_RW_SPACE
   Not enough read/write storage space to load the object module.

LOADER_$NOT_A_PROGRAM
   The name you supplied is not an object module.

LOADER_$PEB_REQUIRED
   Attempted to execute an object that contains instructions specifically for the floating
   point performance enhancement board (PEB), on a machine that does not have the
   board.

LOADER_$TOO_MANY_SECTIONS
   Too many sections in object module.

LOADER_$TOO_MANY_UNDEFINED
   Too many undefined references in object module.

LOADER_$UNDEF(GLBL
   Made reference to an unresolved global at runtime.

LOADER_$UNDEF.GLBL_IN_LIB
   Library contains unresolved global variables, detected because the
   PM_$NO_UNRESOLVEDS option was set.
LOADER ERRORS

LOADER_$UNIX_INIT_REQUIRED
C library initialization required; no C library is installed. /LIB/CLIB must always be present to run a C program. In practice, you should never see this error, since you cannot boot a node without installing CLIB.

LOADER_$WRONG_VERSION
The LOADER could not understand the object module format because the version differed.
LOGIN DATA TYPES

CONSTANTS

NAME_$PNAMLEN_MAX 256 Maximum length of a pathname.

DATA TYPES

NAME_$PNAME_T An array of up to NAME_$PNAMLEN_MAX (256) characters.

LOGIN_$STRING_T An array of up to 256 characters.

LOGIN_$OPT_T A 2-byte integer. Options for LOGIN_$LOGIN system call. One of the following pre-defined values:

LOGIN_$LOCAL_CONN
Reserved.

LOGIN_$LOG_EVENTS
Indicates that you are supplying a procedure to record user's attempts to log in.

LOGIN_$OPT_SET_T A 4-byte integer. A set of LOGIN options in LOGIN_$OPT_T format. For a list of options, see LOGIN_$OPT_T above.

LOGIN_$PROC_REC_T A record of pointers to LOGIN functions and procedures that you supply to LOGIN_$LOGIN. The diagram below illustrates the LOGIN_$PROC_REC_T data type:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0:</td>
</tr>
<tr>
<td>Integer</td>
<td>4:</td>
</tr>
<tr>
<td>Integer</td>
<td>8:</td>
</tr>
<tr>
<td>Integer</td>
<td>12:</td>
</tr>
</tbody>
</table>

Field Description:
PREAD
Pointer to read function that you supply to LOGIN_.SLOGIN.

PWRITE
Pointer to write procedure that you supply to LOGIN_.SLOGIN.

HELP
Pointer to help procedure that you supply to LOGIN_.SLOGIN.

OPEN_LOG
Pointer to open log function that you supply to LOGIN_.SLOGIN.

STREAM_.TID_.T
A 2-byte integer. Open stream identifier.

STATUS_.ST
A status code. The diagram below illustrates the STATUS_.ST data type:

Field Description:

ALL
All 32 bits in the status code.

FAIL
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

SUBSYS
The subsystem that encountered the error (bits 24 - 30).
LOGIN DATA TYPES

MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).

UNIV_PTR
A 4-byte integer. A pointer to allocated storage.
LOGIN_$CHHDIR

Changes the home directory that is listed in the registry ACCOUNT file.

FORMAT
LOGIN_$CHHDIR (login_ptr, home_dir, home_dir_len, status )

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

home_dir
Name of the supplied home directory. This is a UNIV character string.

home_dir_len
Number of characters in "home_dir." This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE

Use this call to change a user's home directory listed in the network registry account file. Before using this call, you must use LOGIN_$OPEN to initialize the LOGIN manager, and LOGIN_$SET_PPO to set the PPO (person, project, organization) files to the user whose home directory you want to change. Use LOGIN_$LOGIN when you are finished changing registry ACCOUNT files.
LOGIN_$CHPASS

LOGIN_$CHPASS

Change the password that is listed in the registry account file.

FORMAT
LOGIN_$CHPASS (login_ptr, password, pass_len, status)

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

password
Name of the supplied password. This is a UNIV character string.

pass_len
Number of characters in "password." This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE

Use this call to change the user's password in the registry account file. Before using this call, you must use LOGIN_$OPEN to initialize the LOGIN manager, and LOGIN_$SET_PPO to set the PPO (person, project, organization) files to the user whose password you want to change. Use LOGIN_$LOGIN when you are finished changing registry ACCOUNT files.
LOGIN_$CKPASS

Checks the supplied password against the one listed in the registry ACCOUNT file.

FORMAT
LOGIN_$CKPASS (login_ptr, password, pass_len, status )

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes, in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

password
Name of the supplied password. This is a UNIV character string.

pass_len
Number of characters in "password." This is a 2-byte integer.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE
Use this call to check the supplied password against the password listed in the registry ACCOUNT file. Before using this call, you must use LOGIN_$OPEN to initialize the LOGIN manager, and LOGIN_$SET_PPO to set the PPO (PERSON, PROJECT, ORGANIZATION) files to the user whose password you want to check. Use LOGIN_$LOGIN when you are finished changing registry ACCOUNT files.
LOGIN_$CLOSE

LOGIN_$CLOSE
Closes a LOGIN operation.

FORMAT
LOGIN_$CLOSE (login_ptr, status )

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE
Use this call to release the resources that the LOGIN manager used for LOGIN operations. The LOGIN manager updates the account files at this time. If a previous LOGIN call had difficulty with an account file, a LOGIN_$CLOSE can fail, which means that the registry will not be updated accurately. You will have to repeat the entire sequence since you opened LOGIN with LOGIN_$LOGIN. Call this routine even if the LOGIN_$OPEN fails.
LOGIN_$ERR_CONTEXT

Locates the file that failed during LOGIN operation.

FORMAT

LOGIN_$ERR_CONTEXT (login_ptr, err_status, bad_name, name_len, status )

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes, in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

OUTPUT PARAMETERS

err_status
Error status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

bad_name
Pathname of the file that failed, in NAME_$PNAME_T format. This is an array of up to 256 characters.

name_len
Number of characters in "bad_name." This is a 2-byte integer.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE

Use this call to locate the pathname of the PPO file failed during a LOGIN operation.
LOGIN

LOGIN

Allows a user to log in to a program.

FORMAT

ok := LOGIN_$LOGIN (stream, login_opt, login_procedures, status)

RETURN VALUE

ok

Returns TRUE if LOGIN_$LOGIN is successful, returns FALSE if not.

INPUT PARAMETERS

stream
Number of the stream where the user will log in, in STREAM_$ID_T format. This is a 2-byte integer.

login_opt
Option you can specify in your log in procedure, in LOGIN_$OPT_SET_T format. This is a 2-byte integer. Specify the predefined value:

LOGIN_$LOG_EVENTS
Tells LOGIN_$LOGIN that you are supplying a procedure to record log in events.

login_procedures
Pointers to input/output routines that you supply to LOGIN_$LOGIN, in LOGIN$_$PROC_REC_T format.

The following are the procedures and functions you supply to LOGIN_$LOGIN with their input and output parameters.

NOTE: The INPUT parameters are the parameters that LOGIN_$LOGIN supplies to the routines. The OUTPUT parameters are the parameters that the routines pass to LOGIN_$LOGIN.

YOUR_READ
Reads input line and passes it to LOGIN_$LOGIN.

Format

int := YOUR_READ(stream, inbuf, inlen, pstr, plen, echo, fillbuf, fillbuflen)

Return Value

int

Returns an integer indicating the length of the message in "inbuf."

Input Parameters

stream
Number of stream associated with input, in STREAM_$ID_T format. This is usually STREAM_$STDIN.
inlen  Maximum length of "inbuf." This is a 2-byte integer.
pstr  Prompt string in LOGIN_$STRING_T format. This is a UNIV character array of 256 characters.
plen  Length of "pstr." This is a 2-byte integer.
echo  Indicates whether the input should be echoed. LOGIN_$LOGIN returns TRUE when prompting for a PPO, FALSE when prompting for a password.
fillbuf  Pre-fill buffer with string in LOGIN_$STRING_T format. This is a UNIV character array of 256 characters. Do not use this parameter, as it is specific to the Display Manager.
fillbufen  Length of "fillbuf." This is a 2-byte integer. Do not use this parameter, as it is specific to the Display Manager.

Output Parameters

inbuf  Login string that YOUR_READ passes to LOGIN_$LOGIN, in LOGIN_$STRING_T format. This is a UNIV character array of 256 characters.

YOUR_WRITE  Writes error and help messages to output.

Format

YOUR_WRITE (stream, pstr, plen)

Input Parameters

stream  Number of stream associated with output, in STREAM_$ID_T format. This is usually STREAM_$STDOUT.
pstr  Message to output, in LOGIN_$STRING_T format. This is a UNIV character array of 256 characters.
plen  Length of "pstr." This is a 2-byte integer.

YOUR_HELP  Provides help message that LOGIN_$LOGIN supplies when user types h[elp] at the prompt.

Format

YOUR_HELP (stream)

Input Parameters

stream  Number of stream on which to output help procedure, in STREAM_$ID_T format.
YOUR_OPEN_LOG
Records log in events in a log file.

Format

ok := YOUR_OPEN_LOG (log_file, log_file_len, log_stream)

Return Value

ok Returns TRUE if YOUR_OPEN_LOG opened successfully, returns FALSE if not.

Input Parameters

log_file Name of your log file, in LOGIN_$STRING_T format. This is a UNIV character array of 256 characters.

log_file_len Length of "log_file." This is a 2-byte integer.

Output Parameters

log_stream Number of the stream associated with "log_file," in STREAM_$ID_T format.

OUTPUT PARAMETERS

status Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE

Use this call to write a log-in procedure that allows you to limit user's access to a specified program. LOGIN_$LOGIN does not handle input/output directly. Rather, you supply your own input/output routines so you can handle the special input/output needs of your application. The arguments of these routines must correspond to the parameters listed above.

To supply LOGIN_$LOGIN with the addresses of your input/output procedures, you must compile them in a separate module.

When the user supplies a PPO and password to LOGIN_$LOGIN, the system call checks the supplied password against the password listed in the registry. If the passwords match, LOGIN_$LOGIN will log the user in.

This system call corresponds to the Shell LOGIN command.
LOGIN_$OPEN

Initializes the LOGIN manager.

FORMAT
LOGIN_$OPEN (mode, login_ptr, status)

INPUT PARAMETERS
mode
Mode of LOGIN operation, in LOGIN_$MODE_T format. This is a 2-byte integer. Specify one of the following predefined values:

LOGIN_$READ Initializes LOGIN in read access mode only. Allows user to view registry account files only.

LOGIN_$UPDATE Initializes LOGIN in read/write access mode. Allows user to view and change registry account files.

OUTPUT PARAMETERS
login_ptr
Pointer to internal LOGIN datatypes in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE
LOGIN_$OPEN sets up storage for internal LOGIN datatypes. Use this call before using the following LOGIN system calls:

• LOGIN_$CHKPASS
• LOGIN_$CHPASS
• LOGIN_$CHHDIR

Even if LOGIN_$OPEN fails, you must also call LOGIN_$CLOSE.
LOGIN_$SET_PPO

LOGIN_$SET_PPO

Sets the PPO (person, project, organization) files of the user whose account file you want to check or change.

FORMAT
LOGIN_$SET_PPO (login_ptr, ppo, ppo_len, status )

INPUT PARAMETERS

login_ptr
Pointer to internal LOGIN datatypes, in LOGIN_$PTR format. This is a UNIV_PTR data type, and is 4 bytes long.

ppo
Name of user's PPO file. This is a UNIV character string.

ppo_len
Number of characters in "ppo." This is a 2-byte integer. If "ppo_len" is zero, LOGIN_$SET_PPO uses the PPO files of the user who is currently logged in.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the LOGIN Data Types section for more information.

USAGE
Use this call to set the PPO files before checking or making any changes to the registry. If you do not call this routine explicitly, the LOGIN manager automatically sets the PPO files to the user who is currently logged in.
LOGIN ERRORS

ERRORS

LOGIN$_BAD_PASSWD
    User supplied an invalid password, detected by LOGIN$_CKPASS.

LOGIN$_ERR_EXIT
    User wants the call to exit.

LOGIN$_ERR_SHUT
    User wants the caller to shut down.

LOGIN$_NO_ROOM
    Not enough storage to initialize LOGIN's internal datatypes.
**CONSTANTS**

NAME_$_COMPLEN_MAX  32  Maximum length of a name.

**DATA TYPES**

NAME_$_NAME_T  An array of up to NAME_$_COMPLEN_MAX (32) characters.

UID_$_T  A process UID. The diagram below illustrates the UID_$_T data type:

Field Description:

HIGH
The high four bytes of the UID.

LOW
The low four bytes of the UID.

STATUS_$_T  A status code. The diagram below illustrates the STATUS_$_T data type:

Field Description:

FAIL
subsys
subsys
mode

Apollo Confidential
Field Description:

ALL
All 32 bits in the status code.

FAIL
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

SUBSYS
The subsystem that encountered the error (bits 24 - 30).

MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).
PM_$GET_HOME_TXT

PM_$GET_HOME_TXT

  Returns the home directory of the calling process as a string.

FORMAT

PM_$GET_HOME_TXT (maxlen, home, len)

INPUT PARAMETERS

maxlen

  Maximum number of characters to be returned (at most, the size of the buffer you assign for home). This is a 2-byte positive integer. This parameter need not exceed 256.

OUTPUT PARAMETERS

home

  Pathname of the home directory for the SID (log-in identifier) of this process. This is an array of up to 256 characters.

len

  Number of characters returned in the home parameter. This is a 2-byte positive integer.

USAGE

  The home directory is obtained from the network registry when you log in and is inherited by all your processes.
PM\_\$GET\_SID\_TXT

Returns the SID (log-in identifier) of the calling process as a string.

**FORMAT**

PM\_\$GET\_SID\_TXT (maxlen, sid, len)

**INPUT PARAMETERS**

*maxlen*

Maximum number of characters to be returned (at most, the size of the buffer you assign for home). This is a 2-byte positive integer. This parameter need not exceed 140.

**OUTPUT PARAMETERS**

*sid*

String containing the person, project, organization and node ID of the SID (log-in identifier) of this process, in the form:

person.group.project.nodeid

This is an array of up to 140 characters.

*len*

Number of characters returned in the log-in identifier. This is a 2-byte positive integer.

**USAGE**

Your SID is the full identifier obtained from the network registry when you log in and is inherited by all your processes.
PM_$SET_NAME

Assigns a name to a given process UID.

FORMAT

PM_$SET_NAME(name, length, process_uid, status)

INPUT PARAMETERS

name
The name you want to give to the process in NAME_$NAME_T format. This is an array of up to 32 characters.

length
Number of characters in "name." This is a 2-byte integer.

process_uid
The UID of the process that you want to name in UID_$T format. This data type is 8 bytes long. See the PM Data Types section for more information.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PM Data Types for more information.

USAGE

Use this call to assign a name to a given process. Once a process has a name, you cannot rename or remove the name from the process.
ERRORS

PM_$ALREADY_NAMED

Attempted to assign a name to a process UID that was already named. You cannot change the name of a process already named by the Shell CRP command, or the DM commands, CP, CPO, or CPS.

PM_$NOT_FOUND

The process manager cannot find the process specified in the call.
DATA TYPES

PROC2-$INFO_T

Total Size: 36

Predefined Record:
uid_st

proc_$state_t

time_$clock_t

Field Description:

STACK_UID
UID of user stack.

STACK_BASE
Base address of user stack.

STATE
Process state - ready, waiting, etc.

USR
User status register.

UPC
User program counter.

USP
User stack pointer.

USB
User stack base pointer (A6).
CPU TOTAL
Cumulative cpu time used by process.

PRIORITY
Process priority.

PROC2_UID_LIST_T
An array of UIDs (in UID_$T format) of up to 24 elements.

PROC2_STATE_T
A 2-byte integer. State of a user process. Any combination of the following pre-defined values:

PROC2_waiting
Process is waiting.

PROC2_suspended
Process is suspended.

PROC2_suspended_pending
Process suspension is pending.

PROC2_bound
Process is bound.

UID_$T
A type UID. The diagram below illustrates the UID_$T data type:

<table>
<thead>
<tr>
<th>Field Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH</td>
</tr>
<tr>
<td>The high four bytes of the UID.</td>
</tr>
<tr>
<td>LOW</td>
</tr>
<tr>
<td>The low four bytes of the UID.</td>
</tr>
</tbody>
</table>

STATUS_$T
A status code. The diagram below illustrates the STATUS_$T data type:

Field Description:

HIGH
The high four bytes of the UID.

LOW
The low four bytes of the UID.
Field Description:

ALL
All 32 bits in the status code.

FAIL
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

SUBSYS
The subsystem that encountered the error (bits 24 - 30).

MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).
PROC2_$GET_INFO

Returns information about a process.

FORMAT

PROC2_GET_INFO (process-uid, info, info-buf-length, status)

INPUT PARAMETERS

process-uid

The UID of the process for which you want information, in UID_$T format. This data type is 8 bytes long. See the PROC2 Data Types section for more information.

You can get process UIDs by calling PROC2_$WHO_AM_I and PROC2_$LIST.

If the process-uid in the call is the caller's own process, the only information returned is the stack UID and virtual address. If you want to find out the amount of CPU time used by the caller's process, use PROC1_$CPU_TIME.

info-buf-length

Length of the information buffer allotted for returned information, in bytes. This is normally 36 bytes.

OUTPUT PARAMETERS

info

Information about the process, in PROC2_INFO_T format. This data type is 36 bytes long. See the PROC2 Data Types section for more information.

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PROC2 Data Types section for more information. Possible values are:

STATUS_$OK Completed successfully.

PROC2_$IS_CURRENT

Specified calling process UID (success).

PROC2_$UID_NOT_FOUND

Specified UID is not on node.

USAGE

GET_$INFO returns information about a process when supplied with a process UID. The information returned consists of the following:
• The program state (ready, waiting, suspended, SUSP_PENDING, bound).
• The User Status Register (USR).
• The User Program Counter (UPC).
• The user stack pointer (A7).
• The stack base pointer (A6).
• The amount of CPU time used.
• The CPU scheduling priority.
PROC2_ $LIST

Returns a list of existing level 2 (user) processes on the caller's node.

FORMAT
PROC2_ $LIST (uid-list, max-num-u1ds, number-u1ds)

OUTPUT PARAMETERS

uid-list
The UIDs of the active level 2 processes on the system, in PROC2_ $UID_ LIST_ T format. This is a 24-element array of UIDs. Each UID is a 4-byte integer in UID_ $T format.

INPUT PARAMETERS

max-num-u1ds
Maximum number of process UIDs to be returned. (At most, the size of the buffer you assign for "uid-list." This is a 2-byte integer.

OUTPUT PARAMETERS

number-u1ds
Number of active level 2 processes on the node, even if that number is greater than "max-num-u1ds." This is a 2-byte integer.

USAGE
The UIDs of all level 2 processes (user processes) on the caller's node, up to "max-num-u1ds," are returned.
PROC2_$SET_PRIORITY

PROC2_$SET_PRIORITY

Sets the priority of a process.

FORMAT

PROC2_$SET_PRIORITY( process_uid, low, high, status)

INPUT PARAMETERS

process_uid

The UID of the process that you want to change the priority level, in UID_$T format. This data type is 8 bytes long. See the PROC2 Data Types section for more information.

low

The lower boundary of the priority. This is a 2-byte integer within the range of 1 to 16. The default lower boundary is 3.

high

The higher boundary of the priority. This is a 2-byte integer within the range of 1 to 16. The default higher boundary is 14

OUTPUT PARAMETERS

status

Completion status, in STATUS_$T format. This data type is 4 bytes long. See the PROC2 Data Types section for more information.

USAGE

Use this call to change the priority of a program. The process priority is an integer ranging from 1 (low) to 16 (high). The Display Manager runs at a priority of (16,16). When the operating system decides which process to run next, it chooses the process that currently has the highest priority.

The priority changes while a process executes. Its priority increases as the process waits for events, and decreases as it computes for long periods without waiting.
PROC2_$WHO_AM_I

Returns the UID of the calling process.

FORMAT
PROC2_$WHO_AM_I (my-uid)

OUTPUT PARAMETERS
my-uid
  The UID of the calling process, in UID_$T format. This data type is 8 bytes long. See the
  PROC2 Data Types section for more information.

USAGE
  You can use a UID obtained through this call to find out information about your process
  from the PROC2_$GET_INFO call.
PROC2 ERRORS

ERRORS

PROC2 _$BAD_STACK_BASE
  Bad stack base.

PROC2 _$IS_CURRENT
  Request is for current process.

PROC2 _$UID_NOT_FOUND
  UID of given process not found.

PROC2 _$UID_NOT_LEVEL_2
  Not a level 2 process.

STATUS_$OK
  Successful completion.
RWS DATA TYPES

DATA TYPES

RWS_POOL_T

A 2-byte integer. Types of pools to allocate read/write or heap storage. One of the following pre-defined values:

- **RWS_STD_POOL**
  Standard pool makes storage accessible to calling process only.

- **RWS_STREAM_TM_POOL**
  Stream pool makes storage accessible to calling program and to a program invoked with the UNIX EXEC system call.

- **RWS_GLOBAL_POOL**
  Global pool makes storage accessible to all processes.

STATUS_T

A status code. The diagram below illustrates the STATUS_T data type:

![Status Code Diagram]

Field Description:

**ALL**
All 32 bits in the status code.

**FAIL**
The fail bit. If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module (bit 31).

**SUBSYS**
The subsystem that encountered the error (bits 24 - 30).
MODC
The module that encountered the error (bits 16 - 23).

CODE
A signed number that identifies the type of error that occurred (bits 0 - 15).

UNIV_PTR
A 4-byte integer. A pointer to allocated storage.
RWS_$ALLOC

Allocates read/write storage for FORTRAN or Pascal programs.

FORMAT
RWS_$ALLOC (nbytes, pointer)

INPUT PARAMETERS
nbytes
The number of bytes of storage needed. This is a 4-byte integer.

OUTPUT PARAMETERS
pointer
The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero means that RWS_$ALLOC could not allocate the desired storage.

USAGE
RWS_$ALLOC allocates the specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

This routine is useful for allocating different quantities of dynamic storage, depending on a run-time factor.
RWS_$ALLOC_HEAP

Allocates heap storage for programs.

FORMAT

pointer = RWS_$ALLOC_HEAP (nbytes)

RETURN VALUE

pointer

The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero means that RWS_$ALLOC_HEAP could not allocate the desired storage.

INPUT PARAMETERS

nbytes

The number of bytes of storage needed. This is a 4-byte integer.

USAGE

RWS_$ALLOC_HEAP allocates the specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

This routine is useful for allocating different quantities of dynamic storage, depending on a run-time factor.
RWS\_\$ALLOC\_HEAP\_POOL

RWS\_\$ALLOC\_HEAP\_POOL

Allocates heap storage in a specified pool.

**FORMAT**

pointer = RWS\_\$ALLOC\_HEAP\_POOL(aloc\_pool, nbytes )

**RETURN VALUE**

pointer

The address of the new storage space, in UNIV\_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS\_\$ALLOC\_HEAP\_POOL could not allocate the desired storage.

**INPUT PARAMETERS**

**alloc\_pool**

Pool where storage will be allocated, in RWS\_\$POOL\_T format. This is a 2-byte integer. Specify one of the following following predefined values:

RWS\_\$GLOBAL\_POOL

Global pool makes storage accessible to all processes.

RWS\_\$STD\_POOL

Standard pool makes storage accessible to calling program only.

RWS\_\$STREAM\_TM\_POOL

Stream pool makes storage accessible to calling program and to a program invoked with a UNIX EXEC system call.

**nbytes**

Number of bytes of storage needed. This is a 4-byte integer.

**USAGE**

RWS\_\$ALLOC\_HEAP\_POOL allocates a specified number of bytes of heap storage to the calling process and returns the address of the storage area.

Use this call when you want to control storage access. You can specify that the storage be accessed by the calling process only, by the calling program and a program invoked with a UNIX EXEC system call, or by all programs.

Due to a current limitation, you cannot use this call in FORTRAN programs due to FORTRAN calling conventions. This will be corrected in the next AEGIS Software Release.
RWS\_\$ALLOC\_RW

Allocates read/write storage for Pascal programs.

**FORMAT**

\[\text{pointer} = \text{RWS}\_\$\text{ALLOC}\_\text{RW}(\text{nbytes})\]

**RETURN VALUE**

\[\text{pointer}\]

The address of the new storage space, in UNIV\_PTR format. This is a 4-byte integer. A returned address of zero means that RWS\_\$ALLOC\_RW could not allocate the desired storage.

**INPUT PARAMETERS**

\[\text{nbytes}\]

The number of bytes of storage needed. This is a 4-byte integer.

**USAGE**

RWS\_\$ALLOC\_RW allocates the specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

This routine is useful for allocating different quantities of dynamic storage, depending on a run-time factor.
RWS_$ALLOC_RW_POOL

RWS_$ALLOC_RW_POOL

Allocates read/write storage in a specified pool.

FORMAT

pointer = RWS_$ALLOC_RW_POOL(alloc_pool, nbytes)

RETURN VALUE

pointer

The address of the new storage space, in UNIV_PTR format. This is a 4-byte integer. A returned address of zero (NIL) means that RWS_$ALLOC_RW_POOL could not allocate the desired storage.

INPUT PARAMETERS

alloc_pool

Pool where storage will be allocated, in RWS_$POOL_T format. This is a 2-byte integer. Specify one of the following following predefined values:

RWS_$GLOBAL_POOL

Global pool makes storage accessible to all processes.

RWS_$STD_POOL

Standard pool makes storage accessible to calling program only.

RWS_$STREAM_TM_POOL

Stream pool makes storage accessible to calling program and to a program invoked with a UNIX EXEC system call.

nbytes

Number of bytes of storage needed. This is a 4-byte integer.

USAGE

RWS_$ALLOC_RW_POOL allocates a specified number of bytes of read/write storage to the calling process and returns the address of the storage area.

Use this call when you want to control storage access. You can specify that the storage be accessed by the calling process only, by the calling program and a program invoked with a UNIX EXEC system call, or by all programs.

Due to a current limitation, you cannot use this call in FORTRAN programs due to FORTRAN calling conventions. This will be corrected in the next AEGIS Software Release.
RWS_$RELEASE_HEAP

Releases storage allocated using the RWS_$ALLOC_HEAP call.

FORMAT
RWS_$RELEASE_HEAP

INPUT PARAMETERS

pointer
The address heap storage space, in UNIV_PTR format. This is a 4-byte integer.

This must be a pointer returned by a call to RWS_$ALLOC_HEAP.

OUTPUT PARAMETERS

status
Completion status, in STATUS_$T format. This data type is 4 bytes long. See the RWS Data Types section for more information.

USAGE

RWS_$RELEASE_HEAP is used in conjunction with the RWS_$ALLOC_HEAP call. RWS_$ALLOC_HEAP dynamically allocates storage for a program, returning a pointer to the new storage. When you no longer need the storage, you release it by passing the pointer to RWS_$RELEASE_HEAP.
RWS ERRORS

ERRORS

RWS_$LEVEL_FAILURE
User program wrote over the storage where the system stored the program level information.

RWS_$NOT_HEAP_ENTRY
Argument to RWS_$RELEASE_HEAP did not refer to storage allocated with RWS_$ALLOC_HEAP.

RWS_$SCRIBBLED_OVER
User program wrote over the storage where the system stored the heap's process information.

RWS_$WRONG_LEVEL
Attempted to release storage that was allocated by a program at a superior (lower) program level. This error can occur when using RWS_$STD_POOL or RWS_$STREAM_TM_POOL.
Index

'NODE_DATA/SIOLOGIN_LOG 3-3
ACCOUNT file
  changing with LOGIN calls 3-1
ACL
  tables for, in FU system calls 2-6
Argument
  definition 1-2
BAF system calls
  handling dynamic storage with 5-1
  improving performance 5-5
  insert files 5-1
  overview of 5-4
  using in proper sequence 5-4
BAF_$ADD_SPACE 5-4
BAF_$ALLOC 5-4
BAF_$CREATE 5-4
BAF_$FREE 5-4
BAF_$NO_ROOM 5-6
BAF_$SHARED 5-4
Binding
  LOGIN bin files 3-1
Brackets
  ignoring on command line 1-7
Breakpoints
  setting in copy of object module 4-2
C
  storage allocation in 5-2
CAL_$DECODE_ASCII_DATE 2-7
CAL_$DECODE_ASCII_TIME 2-7
CAL_$DECODE_LOCAL_TIME 3-11
CAL_$ENCODE_TIME 2-7
CAL_$REMOVE_LOCAL_OFFSET 2-7
Changing
  registry account files 3-12
CL Options 1-5
  adding to current set 1-7
  obsolete 1-7
  removing from current set 1-7
CL system calls
  abbreviated options 1-11
CL options 1-5
  error on invalid flags 1-12
  errors during 1-4
  getting arguments 1-14
  getting next token in list 1-14
  initializing CL 1-4
  initializing, examples 1-8
  insert files 1-1
  miscellaneous 1-17
  obsolete CL options 1-7
  parsing file 1-4
  parsing the command line 1-2
  parsing user input 1-4
  sample program 1-18
  using in proper sequence 1-2
  verifying names 1-6
CL_$CHECK_FLAG 1-11
CL_$CHECK_UNCLAIMED 1-3, 1-12
CL_$COMMENTS 1-7
CL_$DASH_DFT_NOP 1-7
CL_$DASH_NAMES 1-6
CL_$DASH_NOP 1-6
CL_$FIRST 1-14
CL_$GET_ARG 1-13, 2-7, 2-10, 3-18
CL_$GET_ARGS 1-3
CL_$GET_DERIVED_NAME 1-15
CL_$GET_ENUM_FLAG 1-11
CL_$GET_FLAG 1-11, 2-2, 2-5, 4-5
CL_$GET_FLAG_INFO 1-12, 1-17
CL_$GET_NAME 1-3, 1-13
CL_$GET_NAME_INFO 1-17
CL_$GET_NUM 1-3, 1-13
CL_$GET_SET 1-17
CL_$INIT 1-3, 1-4
CL_$KEYWORD_DELIM 1-5
CL_$MATCH 1-17, 2-19, 3-18
CL_$MATCH_ERROR 1-6
CL_$NAME_DFT_STDIN 1-7
CL_$NEXT 1-14
CL_$NO_COMMENTS 1-7
CL_$NO_KEYWORDELIM 1-5
CL_$NO_MATCH_OK 1-6
CL_$NO_MATCH_WARNING 1-6
CL_$NO_STAR_NAMES 1-6
CL_$NO_WILDCARDS 1-6
CL_$PARSE_ARGS 1-5
CL_$PARSE_INPUT 1,4, 1-10, 4-5
    program example 1-10
CL_$PARSE_LINE 1-4
    program example 1-9
CL_$REREAD 1-17
CL_$REREAD_FLAGS 1-17
CL_$REREAD_NAMES 1-17
CL_$RESET_OPTIONS 1-5, 1-7
CL_$SET_DERIVED_COUNT 1-5
CL_$SET_NAME_PREFIX 1-5
CL_$SET_OPTIONS 1-5, 1-7
CL_$SET_STREAMS 1-17
CL_$SET_VERB 1-17
CL_$SET_WILD_OPTIONS 1-5, 1-8
CL_$SETUP 1-4, 2-18
CL_$STAR_NAMES 1-6
CL_$VERIFY 1-17
CL_$VERIFY_ALL 1-6
CL_$VERIFY_NONE 1-6
CL_$VERIFY_WILD 1-6
CL_$WILDCARDS 1-6

Command line
   CL system calls to read 1-3
   how CL parses 1-2
   ignoring brackets 1-7
   initializing 1-3
   model for parsing, figure 1-3
   parsing 1-1
   reading tokens from 1-1
Command Line Handler
   See also CL system calls
Comparing
   source tree to a target tree 2-2
Copying
   file, example 2-8
   files and trees 2-2
CP (create_process) 4-8
CPO (create_process_only) 4-8
CPS (create_process_server) 4-8
CRP (create_a_process) 4-8

Date
   appending to a file 2-2
Deleting
   files and trees 2-2
Derived names 1-7
   definition 1-2
   getting, command line 1-16
   getting, program example 1-16
   handling 1-2
DM commands
   CP (create_process) 4-6
   CPO (create_process_only) 4-6
   CPS (create_process_server) 4-6
DTM/DTU
   preserving 2-8
Dynamic storage
   handling 5-1
Error reporting
   unresolved global variables 4-2
Errors
   CL 1-4
Example
   returning 16-bit value with
      PM_$CALL 4-3
File and Tree Utility
   See also FU system calls
Flags
   checking for CL 1-11
   definition 1-1
   getting arguments associated with,
      program example 1-18
   lowercase letters for defining 1-11
   specifying synonymous flags 1-11
   synonymous, example 1-12
FORTRAN
   allocating storage 5-2
   storage limitation 5-2
FU options
   providing Shell Command options with
      2-3
   setting, example 2-5
FU system calls
   controlling operations by setting
      options 2-2
corresponding Shell commands 2-2
insert files 2-1
overview 2-1
releasing storage 2-6
using in proper sequence 2-1

Handling
dynamic storage 5-1
files and trees with FU system calls 2-1
files, trees, FU example 2-6

Heap
combining adjacent free items 5-5
creating with BAF 5-1
free list 5-5
overhead 5-2
storage for initializing LOGIN 3-13

I/O routines
for LOGIN$_$LOGIN 3-2

Insert files
BAF system calls 5-1
CL system calls 1-1
FU system calls 2-1
loader (PM) system calls 4-1
LOGIN system calls 3-1
PM system calls 4-1
PROC2 system calls 4-1
RWS system calls 5-1

Installing
a library with LOADER system calls 4-1

Invoking
process with /com/shell 3-7

Keywords
definition 1-1

Library
installing 4-2
installing with LOADER system calls 4-1

LOADER system calls
controlling load operation with options 4-2
converting object module to executable format 4-1
difference between PGM$_$INVOKE 4-1
installing a library with 4-1

Loading
and calling a program 4-1

Global sections
loading library with 4-2

Global variables
reporting error on unresolved 4-2
library containing global sections 4-2
Loading object module
See also LOADER system calls
Local storage access
limiting access within calling process 5-3
Logging in
See also LOGIN system calls
LOGIN system calls
changing registry 3-1
closing file 3-13
insert files 3-1
overview 3-1
proper sequence for changing account file 3-12
tailoring LOGIN operation with 3-1
See also LOGIN__$LOGIN
LOGIN__CHDIR 3-1, 3-13, 3-16
LOGIN__CHPASS 3-1, 3-13, 3-15
LOGIN__CKPASS 3-1, 3-13, 3-15
LOGIN__CLOSE 3-1, 3-13
LOGIN__ERR_CONTEXT 3-16
LOGIN__LOG_EVENTS 3-2, 3-3, 3-4
LOGIN__LOGIN 3-1, 3-7
example 3-2
examples of 3-4
I/O routines for 3-2
recording all LOGIN attempts 3-3
writing external I/O routines for 3-3
LOGIN__OPEN 3-1, 3-15
LOGIN__READ 3-13
LOGIN__SET_PPO 3-13, 3-15
LOGIN__UPDATE 3-13
MALLOC 5-2
Managing programs
system calls 4-1
Moving
files 2-2
Multiple processes
sharing information among processes 5-3
sharing storage among 5-4
Names=file
definition 1-1
specifying with * 1-6
Naming
a process 4-5
a process once only 4-6
a process, example 4-6
file, appending a date to 2-2
Object module
writing on copy 4-2
writing on original 4-2
Options
checking for user-specified, CL 1-11
command line, definition 1-1
FU 2-2
ignoring user-specified command line
options 1-6
LOADER 4-2
Overhead
BAF heap 5-5
RWS heap 5-2
RWS system calls 5-4
Overlay process
sharing storage with 5-3
PAD__COOKED 3-9
PAD__PUT_REC 3-9
PAD__RAW 3-9
Performance
improving BAF heap 5-5
PGM__GET_PUID 4-8
PGM__INVOKED 3-7, 4-8
alternative to 4-1
PM system calls
insert files 4-1
sample program 4-6
PM__ALREADY_NAMED 4-5
PM__CALL 4-5
returning 16-bit value with 4-3
returning a value using 4-3
PM__COPYPROC 4-2
PM__INSTALL 4-2
PM__INSTALL_SECTIONS 4-2
PM__LOAD 4-5
PM__LOAD_INFO 4-3
PM__LOAD_WRITABLE 4-2
PM__LOADER_OPTS 4-2

Index-4
Pointers

allocating heap for managing Pool

storage, controlling access with PPO file and LOGIN system calls requiring special

and LOGIN system calls

PPO file

and LOGIN system calls

requiring special

PPRI (process_priority)

PROC2 system calls

insert files

sample program

PROC2 SET_PRIORITY

Process assigning a name to naming once

Process priority

Display Manager example setting a

Process UID

Program level executing program at same

Protected subsystem

LOGIN PST (process_status) PST (process_status) PST (process_status) PST (process_status) PST (process_status) PST (process_status) PST (process_status) PST (process_status) PST (process_status)

Recording

log-in attempts

Registry changing with LOGIN calls

Renaming files

Reporting error, unresolved global variables log-in attempts

Reporting errors wildcard expansion

RWS system calls

handling dynamic storage with insert files when to use

RWS _$ALLOC

RWS _$ALLOC_HEAP

RWS _$ALLOC_HEAP_POOL

RWS _$ALLOC_RW

RWS _$ALLOC_RW_POOL

RWS _GLOBAL_POOL

RWS _$RELEASE_HEAP

RWS _$STD_POOL

RWS _$STREAM_TM_POOL

Sample programs
c1_copy_file

fu_handling_files_trees

login_change_pass_dir

login_login

login_procedures

PM_LOAD

pm_proc2_set_name_priority

Shared storage

Shell invoking

Shell command

PST (process_status)

Shell commands

and corresponding FU system calls

CRP (create_a_process)

PPRI (process_priority)

PST (process_status)

SIGP (signal_process)

SIGP (signal_process)

Stack

allocating storage for

Standard input

reading names from

Start address

of loaded program

Storage

allocating for FORTRAN programs

allocating for Pascal programs

allocating in C programs

handling dynamic

maintaining strict control with BAF

releasing
releasing FU 2-8
requesting specific amount 5-1
shared 5-4
specifying where storage comes from
with BAF 5-4
summary of types of allocation 5-3
using a pool to control access 5-2
when to allocate heap 5-2
when to allocate read/write 5-2
STREAM_$CREATE 3-11
STREAM_$GET_REC 3-9
STREAM_$PUT_CHR 3-8

Tailoring
LOGIN operation 3-1
Token
definition 1-1
marked used when read 1-2
referring to used 1-2
Token list
definition 1-2
figure of 1-2
Token pointer
definition 1-2
Token record
definition 1-2
UNIX EXEC system call 5-3
Verifying pathnames 1-6
Wildcard-name
definition 1-2
Wildcards
expanding 1-2, 1-6
how CL expands 1-8
returning verbatim 1-6
Working set
controlling size of 5-4
desirable behavior 5-5
using heap to control 5-2