A GUIDE TO THE COMMAND META LANGUAGE AND THE COMMAND LANGUAGE INTERPRETER

Donald I. Andrews
Beverly R. Boli
Andrew A. Poggio

January 5, 1977

ARC Catalogue Number 28744

TABLE OF CONTENTS

PREFACE

This Guide is designed for tool builders and tool installers who wish to use the Command Meta Language (CML) and Command Language Interpreter (CLI) to develop and implement a Frontend interface for their tools. Included is a complete guide to programming in CML, an explanation of grammar compilation and compaction, a discussion of CLI operation, a description of the CLI-tool interface, and guidelines for setting up User Profile and Help systems. Appendices at the end of the Guide provide detailed information on these topics.

Before using this Guide we suggest you familiarize yourself with AN INTRODUCTION TO THE FRONTEND [1].

OVERVIEW OF THE CML

The Command Meta Language (CML), a high level, formal language, is designed especially for implementing user command languages for interactive systems. Its flexible and straight-forward conventions allow the programmer to create a consistent and coherent user interface across applications programs or "tools".

CML provides the means to easily create, change, and experiment with the user interface to an interactive tool. Commands available to the user and the interaction methodology and techniques used to specify commands are manipulated independently. Changes in command words, command word structure, prompts, and noise words are simply made, usually requiring little more than trivial edits to the CML program.

A CML program is compiled to produce an object code, called a grammar, which is interpreted by the Command Language Interpreter (CLI). The CLI provides the user with well-formed commands and consistent language features for all tools or programs. It is a command-at-a-time system, in which the tool or program does not directly interact with the terminal, but rather receives fully specified commands from the Frontend. Independent of the tool or program used, the methods are the same for specifying commands and their arguments. Prompting and error and status conditions are also presented to the user in the same way, as are the user aid facilities and methods of requesting help.

The CLI also provides a terminal independent interface to tools. Because the CLI handles all terminal interaction, it will present to the tool a small number of virtual terminal classes. Thus once a tool is developed, little attention need be given to the type or particular characteristics of the terminal the end user may choose to employ. For tools that are designed to make extensive use of display terminals, the CLI presents primitives for allocating windows on the display and allows the tool to write, delete, or move items displayed within the windows.

A CML program describes the command words, noise words, selection requests, etc. that make up a command. The CLI code interacts with the user when he enters the subsystem and as he specifies commands. In the process of interacting with the user, the CML code may call one or a number of execution procedures which "do the work." The CLI automatically provides prompting, questionmark, and <CTRL-S> facilities.

PROGRAMMER'S GUIDE TO CML

This section provides a prose description of CML illustrated with examples from CML grammars. A formal definition of the CML syntax is presented in Appendix 6.

Program Structure

The following is a model of a CML program, showing the relationship of its parts and major conventions. Use this as a reference as you read through the programmer's guide.

```
FILE program-identifier   % comments %
  DECLARE VARIABLE
    variable, ...;
  DECLARE GLOBAL
    globalvariable, ...;
  DECLARE CONSTANT
    name = 1, ...;
  DECLARE COMMAND WORD
    "COMMANDWORD" = 1,
    "CW2" = 5  SELECTOR  POINT = point parse function  TYPEIN
    = built-in selection type  ADDRESS = built-in selection
    type;
  DECLARE FUNCTION
    execution function, ...;
  DECLARE PARSEFUNCTION
    parse function, ...;
  SUBSYSTEM internal-name KEYWORD "HERALD"
    INITIALIZATION
      initrule = ...;
    TERMINATION
      termrule = ...;
    REENTRY
      reentrule = ...;
    HELP
      helprule = ...;
    commandword COMMAND =
      "COMMANDWORD"
        cml elements...
        xroutine(parameter, ...);   %call to execution
        function%
        rule = ...;
    END.
  FINISH
```

A CML program begins with a "FILE" statement of the form:

```
FILE name
```

where "name" is the identifier (or name) of the program code. It must be written in lower case letters and numbers, beginning with a lower case letter. The same identifier must not be used in the same program for another purpose.

The program ends with the statement:

```
FINISH
```

Within the program, there should be a series of declarations followed by a single subsystem.

All variables used in the program must be declared. Other values and attributes must also be declared in CML.

Each program represents a subsystem. A subsystem may include one or more commands and optionally may include rules to be performed upon entering, reentering, or leaving the subsystem. A subsystem may also include general rules defined throughout the subsystem.

Rules are sequences of CML elements that specify the interaction between the user and the system, call various kinds of functions, and so on. Rules may be referenced from within rules or commands. When such a reference is made, it is as if the CML elements represented by that rule were inserted at that point in the rule or command. This allows the definition of general interactions that may be of use in a number of commands or points in a command.

Each of these parts of the CML program will be described independently. The CML elements that make up rules will also be described.

Comments may be enclosed in percent signs (%) and inserted anywhere in the program that a space is legal. The compiler will ignore them.

Declarations

Declarations associate attributes and values with identifier names used in CML programs. Many types of names may be used in CML programs: variables, parse functions, command words, Backend functions, and constants.

Subsystems

There is only one subsystem in each CML program (or grammar). The subsystem construct brackets a set of rules or commands (generally a set of related commands that the tool implementer wants to cluster together). Rules containing the reserved word COMMAND are linked together by the compiler to form a command language sybsystem or tool. The subsystem begins with a statement of the form:

```
SUBSYSTEM name KEYWORD "NAME"
```

where name is the internal name of the subsystem (primarily for debugging purposes) and NAME is the name the user must specify to access commands in the subsystem as well as the name that will appear as the subsystem herald. These two names may be the same, but they must otherwise be unique identifiers within the subsystem.

A subsystem ends with the statement:

```
END.
```

Within the subsystem you may have any number of rules. A rule, as described below, will be known throughout the subsystem, but not outside the subsystem.

A rule preceded by the word "COMMAND" will be available as a command in the subsystem. Commands may not be called as rules. A sample "COMMAND" rule follows:

```
COMMAND replace = "REPLACE"
  type _ editentity
  <"at"> dest _ DSEL(type)
  <"by"> source _ LSEL(type)
  CONFIRM
  xreplace (dest, source, FALSE, FALSE);
```

(The first line may also be written: replace COMMAND = "REPLACE".)

This command allows the user to replace items specified in the rule "editentity" at a point he specifies (using the DESTINATION SELECTION (DSEL) routine and storing the value in the variable "dest"), by something he types in (using the LITERAL SELECTION (LSEL) routine and storing the value in "source"). After the user confirms his command, an execution routine named "xreplace" (in the tool Backend) is called.

A rule preceded by the word "INITIALIZATION" is executed whenever the subsystem is entered, for example,

```
  INITIALIZATION example =
   procl();
```

A rule preceded by the word "TERMINATION" is executed whenever the subsystem is left. A HELP rule is executed when an execution function needs Help from the Frontend to complete its task. A rule preceded by the word "REENTRY" is executed whenever the subsystem is reentered. Only one INITIALIZATION, TERMINATION, HELP, or REENTRY rule is permitted per subsystem (i.e., only one of each kind). Each of these rules may be called from within another rule.

Rules

A CML rule is a series of elements, each of which represents one piece of the system action or interaction with the user. The elements will be described below. Except when preceded by the word COMMAND, the name of a rule (defined to be the given series of CML elements) may be used in other rules. When the name of a rule appears in another rule, the CML code which it represents will be executed at that point.

A rule takes the form:

```
name = element1 element2 element3 ... elementN ;
```

where "name" is any unique name (lowercase letters and numbers, beginning with a letter).

Three examples of rules follow. The first two rules are used to define clusters of command words:

```
editentity = textent / structure;

textent = textl / "TEXT" / "LINK" / "NUMBER";
```

In the first example the rule named "editentity" is defined to include either the rule "textent" or "structure". The second rule defines "textent", first by naming another rule, followed by alternative command words.

```
bckrng = var _ xhelpring(param);
```

In the rule "bckrng" the execution function "xhelpring" is called and the argument "param" is passed to it. The value it returns is stored in the variable "var".

Elements are NOT separated by any delimiter character (except by spaces or the source file structure). The entire rule is terminated by a semicolon.

Alternative groups of elements are separated by a slash (/) in the expression. Alternatives typically provide the user a choice, or provide more than one "path" through a grammar. Parentheses should be used to group elements, particularly when alternative logic and nesting of alternatives is involved, for example:

```
name = (element1 / element2 element3) element4 ;
```

This rule may be read as "either element 1 and element 4, or elements 2, 3, and 4". Note that, by use of parentheses, an alternative may include more than one element.

Elements grouped in square brackets are optional. When command words are placed in square brackets, the user has a choice of typing the expression(s) within the square brackets, or the one following it. If the command words in square brackets are chosen, the user will still have to confirm the command afterwards. Here is an example:

```
output COMMAND =
  "OUTPUT" <"to">
    namefil _ NULL
    type _ "PRINTER"
    [ "APPEND" <"to File">
    namfil _ LSEL(#"OLDFILENAME")
    param2 _ TRUE ]
    CONFIRM
    xoutput (type, namfil, WINDOW, param, param2);
```

In this command the user first specifies "Output Printer"; at that point he has the option to choose either the command word "Append" (and type in the name of a file), or to simply confirm the command as it has thus far been specified. (The remainder of the rule calls the execution function "xoutput" and passes CML values to it.)

Most CML elements have a value. This value may be assigned to a CML variable using a left-arrow (_) in the form:

```
variable _ element
```

Names on the left side of an assignment are assumed to be variables. The variable must have been declared, as described below.

Using CML Elements to Construct a Grammar

Writing a CML grammar is simply a process of building up different kinds of CML elements. User interaction is defined by the order of the CML elements in a subsystem grammar. This section presents a complete list of those elements in an order resembling that in which the grammar writer would be likely to use them.

Variables

All variables declared in a grammar will be initialized to NULL when the grammar is entered. There are three variable domains in CML--local, global, and built-in.

Local Variable

Local variables must be declared in the grammar and are only meaningful in that grammar. Local variables are reset to NULL at the end of every command. A local variable may be declared with the statement:

```
DECLARE VARIABLE var ;
   or
DECLARE var ;
```

You may declare any number of variables in a single statement, i.e.:

```
DECLARE VARIABLE var1, var2,... ;
   or
DECLARE var1, var2,... ;
```

where "var" is any unique name (lowercase letters and numbers, beginning with a letter). All variables declared in a grammar will be initialized to NULL when the grammar is entered.

Global Variable

Global variables must be declared in the grammar and are meaningful only for that grammar. They retain their value until reassigned (i.e., they are not set to NULL at the end of the command). A global variable is declared with the following statement:

```
DECLARE GLOBAL glovar1, glovar2, ...;
```

where "glovar1" is any unique name (lower case letters and numbers, beginning with a letter).

Built-in Variable

Built-in variables exist in the CLI and can be used by any grammar. They retain their value until reassigned. They should not be declared. Here is a list of CML built-in variables and their functions:

DISPLAY--True if the user is at a display terminal.

HALFDUPLEX--True if the user is at a half duplex terminal.

LINEATATIME--True if the user is at a line-at-a-time terminal.

TYPEWRITER--True if the user is at a typewriter terminal.

WINDOW--For a display terminal, holds the location of the window in which the cursor was placed at the time of the last Command Accept.

TERMCHAR--Holds the value of the last character that terminated the last TYPEIN or the last CONFIRM.

HELPCODE--This variable is given a numerical value by the externally callable FE function HELP. (See Appendix 7 for more detailed informaton.)

TERMINATORS--This value is supplied by the grammar. Any character or characters assigned to this variable will act as the Command Accept character.

RESULT1 through RESULT8--Hold the values of the corresponding results of the last execution function called. They retain their value until the command ends or another execution function is called.

Variable Types

Variables are also classified by types. Any variable type is legal in any variable domain. The most frequently used types are described below:

string--A string of characters (or "text").

integer--A number or list of numbers.

null--An empty variable.

Boolean--Allows two variables, true or false.

command word--A CML command word. (See "Command Words" below.)

list--Allows a list of values. See below for more explanation.

For a complete listing of all variable types and a discussion of their uses, see FRONTEND SYSTEM DOCUMENTATION [2].

List Variable

The assignment operator :_ may be used to append a new value to a variable instead of the _ operator which replaces the current value with a new value. If a variable previously contained a single value, then a :_ assignment replaces it by a list containing the old value with the new value appended to it. If the variable previously contained a NULL (the value to which all variables are initialized when the grammar is entered), the list will contain only the new value appended to the variable. (Any NULL other than that initially assigned to the variable list will be added to the list when appended to it.)

Subsequent : assignments to that variable would append the new value to the list. Likewise, if a variable previously contained a list, a _ assignment will append the new value to the list. Here is an example.

```
param :_ param2   param :_ param3
```

If the value being appended to the list is itself a list then the operation is one of concatenating the lists rather than appending a single list element that is itself a list. (Additional syntax will be added to allow for this latter case also.) To reset the list, the variable is assigned a NULL.

Constants

Declared Constants

Constants hold one value throughout a grammar. A declared constant must always be an integer value, assigned in the constant declaration as follows:

```
DECLARE CONSTANT

  name = 1, ...;
```

where name may be lower case letters and digits, beginning with a letter.

Built-in Constants

TRUE, FALSE, NULL, and integers (1, 2, 3, ...) are all built-in constants. NULL is especially useful for clearing the value from a variable. TRUE and FALSE are frequently used as arguments passed to execution functions. The following uses are typical for built-in constants.

```
param _ NULL

xopen( namfil, TRUE, FALSE, WINDOW )
```

Command Words

A command word is a word specified as part of a command (e.g., "Insert" or "Word" in the Insert Word command). The appearance of a command word element in a rule means that the user must specify that (or an alternative) command word at that point in the command specification. A declaration may assign a numeric value to a command word, to be passed to an execution procedure that needs to know which command word was chosen by the user.

Declaring Command Words

A command word declaration precedes a list of command word strings and indicates that the named command words are to have the specified integer tokens (numeric values); optionally they may also be used to collect arguments from the user. The syntax is as follows:

```
DECLARE COMMAND WORD "WORD1"=100, "WORD2"=101;
```

All command words should be declared. You may not use command words identical to the names of the Backend execution or CML files, to the name of the subsystem, nor to any variable names.

Integer tokens are optional; they may be left out for very simple uses of command words. The value must be a positive decimal integer, less than or equal to 127. More than one command word may have the same value (unless of course the Backend procedure must distinguish the user's choice between the two).

Declaring command words as SELECTORS is described below under "Selection Routines".

Command Word Recognition

In the CML description, each command word is represented by its full text. The algorithm used to match a user's typed input against any list of alternative command words is known as "recognition." Each individual's command word recognition mode will determine what characters the user must type to specify the command word. This is handled automatically by the CLI.

Command word elements must be upper case words enclosed in double-quotes (""), for example:

```
"SUBSTITUTE"
```

Command words optionally may be followed by one or more qualifiers that modify the recognition process, separated by spaces, and enclosed in exclamation points. The qualifiers are:

L2--second level (when two command words begin with the same letter, some recognition modes differentiate first from second level command words, e.g., second level are preceded by a space. In a grammar a second level command looks like this:

```
create COMMAND =
  "CREATE"!L2! "FILE" ...
```

number--explicit value for the commandword; supercedes any value assigned by a DECLARE COMMAND WORD:

```
change COMMAND =
  "CHANGE"!146! ...
```

Assigning Command Word Values

Command words have values that may be assigned to CML variables, enabling the user's choice to be passed to subsequent routines (e.g., ent _ "CHARACTER"). An example of a rule containing a command word assignment follows:

```
sort COMMAND =
  "SORT"!L2!
    type _ ("BRANCH" / "GROUP" / "PLEX")
    dest _ DSEL (type)
    CONFIRM
    xsort (dest, WINDOW) ;
```

Remember that the variable gets the value of the command words (i.e., "type" is assigned the value of "BRANCH", "GROUP", or "PLEX"). This value will be assigned as above even if the command word is followed by other CML elements, e.g.:

```
ent _ ("CHARACTER" param _ FALSE
/ "WORD" <"at"> param _ LSEL(#"WORD") )
```

'ent' will get the value of the command word CHARACTER or the value of the command word WORD. The appropriate actions will happen after the user chooses the command word.

You may wish to pass this value without forcing the user to type the command word. This value may be assigned by preceding the command word by a pound-sign (#):

```
ent _ #"CHARACTER"
```

The value of "CHARACTER" will be assigned to 'ent' without forcing the user to type CHARACTER.

Variables as Command Words

A variable whose value is a string or a list of strings may be used as a command word. The syntax, as it appears in a rule, is

```
CW: var
```

as illustrated in the following example:

```
toolname _ ( CW : tools / OPTION <"tool name:">
LSEL(#"TEXT") )
```

If the value is a list, each string in the list acts as an alternative command word. Thus, in the above example, if

```
tools _ ("EDITOR", "COMPILER", "MANAGER")
```

the command words EDITOR, COMPILER, and MANAGER would be available as alternatives to the user at the point in the rule in which the variable 'tools' appears.

Feedback

Noise Words

Noise words appear in a command (enclosed in parentheses) to help users understand the purpose of a command or what input is expected of them at some point. Any string of text may be added to the command feedback line by enclosing the quoted text with angle brackets in a rule:

```
<"text of noise words">
```

SHOW() and CLEAR

Used in a rule, SHOW will display the value of a variable to the user. The variable is placed in parentheses following SHOW. (See "Presenting Execution Function Information to the User", below, for more information on when to use SHOW.)

The entire command feedback area (in display mode) can be erased with the CML element CLEAR. The CLEAR function initializes the command feedback mechanisms and clears the command-feedback area.

An example of the use of both elements follows.

```
substl =
  CLEAR  <"new"> SHOW(type) param2 _ LSEL(type)
  CLEAR <"for old"> SHOW(type) param3 _ LSEL(type)...
```

Control Structures

LOOP

This feature has been included to facilitate grammars where it is desirable to implement commands that never terminate (except when the user types the COMMAND DELETE key). The LOOP construct does not use recursion and therefore does not allow the user to backup across iteration boundaries.

An EXIT appearing within the expression being iterated will cause the iteration to terminate and the CML following the LOOP to be evaluated. The user may, of course, type an abort key and cause the iteration to be terminated along with the rest of the command.

Conditionals

Parameters may be tested for TRUEness, FALSEness, NULLness, or for a binarary relationship with some variable or integer with the conditionals "IF" and "IF NOT".

One may write "IF NOT var" and "IF var" to test whether or not a variable is FALSE (or NULL or contains the integer zero) or TRUE (or contains a non-zero integer or a more complex data structure):

```
(IF param3 / IF NOT param3 substl)
```

You should use this construction with some care. To use both constructions to test a variable you must include a slash between them; otherwise only one test will occur and anything following that point will not be done. For example, the following will never display the last set of noise words, "this is not a display", nor will it ever call xroutine:

```
disrule =
  IF DISPLAY <"this is a display"> IF NOT DISPLAY <"this
  is not a display"> xroutine(arg);
```

If DISPLAY is TRUE the first noise words, "this is a display", will be displayed and then the IF NOT DISPLAY will fail causing the rest of the rule to be ignored. If DISPLAY is FALSE, the IF DISPLAY will fail, and nothing more will be done. Here is the proper way to write the rule:

```
disrule =
  (IF DISPLAY <"this is a display">) / (IF NOT DISPLAY
  <"this is not a display">) xroutine(arg);
```

The assignments var _ FALSE and var _ TRUE are useful here. Note that a global or built-in variable maintains its current value until it is replaced by a new assignment.

Selection Routines

Selections are text or addresses pointed out by users on a display terminal or typed from a keyboard. Three kinds of selection routines are built into CML. They are Literal

Selection (LSEL), Destination Selection (DSEL), and Source Selection (SSEL).

Built-in Selections

The Literal Selection accepts a literal typein from the user as well as text pointed out on a display. It further allows an address if the user types an OPTION character first.

A Destination Selection allows the user to select one of several items the tool has presented to him. The user may select by indicating theitem with a pointing device at a display terminal, or by typing characters that the tool will interpret as an address. Take, for example, a tool designed to manipulate textual or graphical representations of data stored in a file. A delete command in the tool would use a DSEL to allow the user to specify which line in the drawing or which word in the text to delete. Thus, when the tool puts the display image on the screen, it would do so using primitives in the Frontend that supply identifiers for elements of the display. When the user points to an object on the screen the identifier for that object would return to the tool.

A Source Selection is similar to a Destination Selection but also allows the user to supply the argument as a literal typein after typing the OPTION character.

Each of these predefined selection routines prompts the user and receives the input. The selection routine must be passed a command word as its argument (character, word, statement, etc.). The selection routine, together with the command word argument, enables the CLI to prompt the user for the appropriate input and pass it to the proper CLI functions.

If more than one selection is necessary (e.g., to specify both ends of a string of text), the CLI will prompt for both automatically. User input will be interpreted correctly according to the command word passed as the argument. (For instance, if a single address or location on a display is input from the user, and the command word passed is "word", both ends of the word will be found from the single selection.)

The command word argument is enclosed in double-quotes and preceded by a pound-sign (#), and is equivalent to the address of the declared value of that command word:

```
DSEL(#"CHARACTER")
```

Preceding the command word with a pound sign means that the user does not have to specify the command word.

Or you may assign the address of the value of a previously selected command word to a CML variable, then pass the content of the variable, e.g.:

```
ent _ "CHARACTER"
DSEL(ent)
```

Selection Declarations

The COMMAND WORD declaration allows the specification of some command words as arguments to LSEL, DSEL, and SSEL. Three methods of selection are possible: TYPEIN, POINT, and ADDRESS. For each selection method, the syntax of the declaration allows the grammar writer to indicate whether the selection will be performed as one of the built-in selection types or a selection parse function will be used to collect it from the user. (Selection parse functions are described in Appendix 5.)

The built-in selection types are TEXT, CHARACTER, WORD, VISIBLE, STRING, FILENAME, INTEGER, PASSWORD, NEWFILENAME, OLDFILENAME, and INVISIBLE.

If the grammar writer wants one built-in selection (and no selection parse functions) to be performed for each selecton type, he only needs to specify the built-in as the SELECTOR. The syntax is of the form:

```
DECLARE COMMAND WORD
  "BLAP" = 5 SELECTOR = BUILT-IN;
```

where BUILT-IN represents any built-in selection type (listed above).

If the grammar writer wishes to specify each selection type independently, the declaration may look like the folowing:

```
DECLARE COMMAND WORD

  "TEXT" = 27 SELECTOR

     POINT = pnttext  TYPEIN = TEXT  ADDRESS = getaddr;
```

When TEXT is passed as an argument to a selection routine, three things can happen depending on the user input. If the user tries to point to text, the selection parse function pnttext will be called to process it. If the user types in the selection, it will be treated as free text. If he types the address of some text then the parse function getaddr will be called to process it. (If the declaration had specified that ADDRESS = TEXT then free text would be collected as the address and it would be marked as an address string of type 27.)

Execution Functions

Execution functions are arbitrary remote processing functions in a tool Backend, usually invoked to carry out all or part of the execution of a command. An execution function may be called at any point by writing the name of some function, followed by a list of zero or more CML parameters in parentheses. For example, notice the last line in the command rule cited earlier:

```
replace COMMAND = "REPLACE"
  type _ editentity
  <"at"> dest _ DSEL(type)
  <"by"> source _ LSEL(type)
  CONFIRM
  xreplace (dest, source, FALSE, FALSE);
```

Only eight arguments can be passed from the CML to an execution function in the tool Backend, and only eight results can be returned from an execution function. Any returned results may be stored in CML variables with a ->. They may be shown to the user, tested, or passed as arguments to other parse or execution functions. Here is an example.

```
xcreate( namfil, WINDOW -> shwrtn, shwstr );
```

The result returned from the execution routine xcreate will be stored in variable shwrtn. If two results are returned, the second return will be stored in shwstr.

All execution functions must be declared. An illustration of a declaration follows:

```
DECLARE FUNCTION

  xinsert, xreplace, xoutproc;
```

Presenting Execution Function Information to the User

The built-in routine SHOW may be used to display the results of an execution function to the user. On a display terminal, the results are displayed in the command feedback window. They will remain on the screen only until a new command is specified. SHOW may be used in several ways, including SHOW(var) or SHOW(x(args)) or SHOW(var _ x(args)). The function x in this case returns the results, the first of which is displayed to the user. The execution function need not concern itself with how this information is presented to the user.

SHOWCONFIRM may also be used to indicate that the user must confirm the fact that he has seen the message before parsing may continue.

The Frontend also makes available an externally callable function to present status, warning, and error messages to the user. This may be invoked by using SHOWSTATUS. On a display terminal the message is displayed in the tty-window, where it will remain even though a new command may be specified.

An error message resembling that presented by SHOWSTATUS may be provided by the Backend. When an execution function issues an ABORT return, it may supply a string that will be presented to the user as an error message. On a display terminal the message is presented in the tty-window.

IN LINE/OUT OF LINE Execution Functions

When a remote call is made, a qualifier in square brackets [] may be used to indicate the name of a rule or to modify the declared IN LINE/OUT OF LINE attribute of the procedure. If a rule is named for an IN LINE call it is assumed to be a HELP rule and will be processed if the called procedure requests it. If the call is OUT OF LINE, however, the specified rule is processed immediately after issuing the call. See Appendix 7 for a discussion of HELP rules.

Results from the called procedure can be assigned to variables through the use of the "->" construct. Such results are always stored in RESULT1 through RESULT8. However, these built-in variables are set to NULL at the end of each command and after issuing each remote call.

Calls to OUT OF LINE Execution Functions

If a function has been declared OUT OF LINE, then one may write a call to x as x[IN LINE](args) and force the call to be done IN LINE. Likewise a call to a normally IN LINE function can be made OUT OF LINE by calling y[OUT OF LINE](args). This is covered more fully in Appendix 7.

Parse Functions

Parse functions (PFs) are routines written in a high-level language that may be invoked from a CML grammar to perform user interface operations not supported by CML. They may be given arguments and return a result that may be referenced in the grammar, e.g., by assigning it to a CML variable. All parse functions must be declared as follows.

```
DECLARE PARSEFUNCTION name1, name2,... ;
```

A parse function acting as a selection function does not have to be declared as a parse function.

Input Recognizers

CONFIRM

The process of command confirmation is represented in CML by a built-in function, CONFIRM. It waits for the user to type a confirmation character. CONFIRM succeeds if COMMAND ACCEPT or REPEAT is typed.

OPTION

When OPTION precedes one or more elements in a rule, the user must type the OPTION character to access them, for example,

```
name = element1 OPTION element2
```

ANSWER

ANSWER allows the user to respond yes or no to a question. It succeeds if the answer is affirmative and fails if it is not.

CML Examples

Example I

```
create COMMAND =     %creates a new file%
  "CREATE"!L2! "FILE"
    namfil _ LSEL(#"NEWFILENAME") CONFIRM
    xcreate( namfil, WINDOW -> shwstr );
```

Example II

```
load COMMAND =
  "LOAD"     % loads a new file %
    ( "FILE"
      namfil _ LSEL(#"OLDFILENAME") CONFIRM
      xopen( namfil, TRUE, FALSE, WINDOW )/
    "PROGRAM"     % loads a new program %
      namfil _ LSEL(#"OLDFILENAME") CONFIRM
      xload(namfil, WINDOW ) );
```

Example III

```
FILE nswlanguage
  DECLARE VARIABLE
    param2, param3, type, dtype, dest;
  DECLARE GLOBAL
    vs, param;
  DECLARE COMMAND WORD
    "SUBSTITUTE" = 123,
    "STATEMENT" = 29  SELECTOR = STRING,
    "BRANCH" = 26  SELECTOR
      POINT = pbranch  TYPEIN = TEXT  ADDRESS = TEXT,
    "PLEX" = 28 SELECTOR
      POINT = pplex  TYPEIN = TEXT  ADDRESS = TEXT,
    "GROUP" = 27 SELECTOR
       POINT = pgroup  TYPEIN = TEXT  ADDRESS = TEXT;
  DECLARE FUNCTION
    xinit, xfterm, xsubstitute;
  DECLARE PARSEFUNCTION
    festoptelnet, viewspecs;
  % COMMON RULES %
    textent = textl / TEXT" / "LINK" / "NUMBER";
    textl = "CHARACTER" / "WORD" / "LINK" / "NUMBER";
    structure = "STATEMENT" / notstatement;
    notstatement = "GROUP" / "BRANCH" / "PLEX";
  SUBSYSTEM base KEYWORD "BASE"
    INITIALIZATION
      initrule =
        IF NOT xinit() festoptelnet();
    TERMINATION
      termrule =
        xfterm();
    substitute  COMMAND =
      "SUBSTITUTE"     % replaces entities defined in textent
      %
        vs _ NULL  param _ NULL
        type _ textent
        <"in">
          [ OPTION <"Filtered:"> vs _ viewspecs() ]
          dtype _ structure
        <"at"> dest _ DSEL(dtype)
        % collect pairs of entities of type type %
          substl
        CONFIRM
        xsubstitute( dest, param, vs );
        substl =
          CLEAR  <"new"> SHOW(type) param2 _ LSEL(type)
          CLEAR <"for old"> SHOW(type) param3 _ LSEL(type)
          param :  param2  param :_ param3
          <"Finished?"> param3 _ ANSWER
          (IF param3 / IF NOT param3 substl);
    END.
  FINISH
```

GRAMMAR COMPILATION AND COMPACTION

Using a Grammar

Once a CML grammar is written, it must be compiled and compacted to transform it into a form that may be used by the CLI. The process is similar for using grammars on both the PDP-1Ø and PDP-11, with differences noted below in "Using grammars on the PDP-11". The following two sections apply specifically to the PDP-1Ø.

Compiling a CML File

The CML compiler is a TENEX program. To use it, run file <arcsubsys>cgcml.sav, specifying input and output file names when requested by the command. The input file should be a text file and the output will be a relocatable binary file. The output file should have the extension ".cml".

The same compiler may be used from NLS to compile an NLS file. Compile the CML source file with the Programs subsystem command Compile File, using <arcsubsys,cgcml,> for the compiler and specifying ".cml" as the extension of the object file. For example, an appropriate FILE statement might look as follows:

```
FILE exec  %<arcsubsys,cgcml,> to <mydir,exec.cml,>%
```

Compacting a CML File

The compiled grammar must then be compacted. Before running the compacter, the parse function data and code files, if either is needed, must be written and compiled into files with extensions ".pfd" and ".pfc", respectively. These files are optional. However, if either one exists and is changed, or if the grammar is changed, the grammar must be compacted again. If the grammar is changed, it must be recompiled before it is recompacted.

To run the compacter from the TENEX EXEC, run file <arcsusys>cgram.sav. The compacter will ask you for the name of the grammar to be compacted; you should type it in with no extension. For the example above, you would type "exec<CR>". The compacter will look for the grammar file with extension ".cml", a parse function data file with extension ".pfd", and a parse function code file with extension ".pfc" in the connected directory. Assuming you are connected to directory mydir, the compacter would look for <mydir>exec.pfd, <mydir>exec.pfc, and <mydir>exec.cml.

If either parse function file is missing, the compacter will issue a warning and continue. If the grammar file is missing, the compacter will issue an error and stop. In general, warnings issued by the compacter will not affect the compaction process and may be ignored while errors will stop the compacter and must be attended to.

If the compaction is successful, the compacter will issue a "SUCCESSFUL COMPACTION" message. It will then attempt to make a compacted grammar file with extension ".cgr". If this is successful, the compacter will issue a "COMPACTED GRAMMAR FILED" message and stop.

Using Grammars on the PDP-11

The process for transforming the grammar into a form for the PDP-11 is analogous to that for the PDP-10. The compilation process is identical. For compaction, <arcsubsys>cgram11 should be used; it expects parse function data and code files with extensions ".pfd11" and ".pfc11", respectively. If the compaction is successful, the compacter will produce a file with the extension ".cgr11".

CLI OPERATION

The CML Grammar

A CML grammar is a series of instructions and associated tables generated by compiling a CML source file with the CML compiler and compacting the compiler output. Examples of instructions are "recognize a command word", "collect a selection", and "call an execution function". The tables contain command word strings, execution function names, and the like. The instructions form a program which the CLI interprets; it is this process of grammar interpretation that produces the high quality user interaction for which the CLI is so well known.

CML Grammar Interpretation

CLI grammar interpretation begins at the top level command in the grammar and is directed by user input and results obtained from processing instructions. Most typically, the user specifies a command word that directs the CLI to interpret a particular command out of all those available in the grammar. The process of grammar interpretation continues until the end of a command is reached. At this point, the CLI cleans up after the command and then repeats the interpretation, beginning again at the top level commands.

CLI User Prompts

User prompts are symbols printed by the CLI at various points during grammar interpretation. They indicate what the input the CLI expects from the user. These prompts are currently used:

C--command word

B--BUG (pointing to something)

T--TYPEIN

A--ADDRESS

OK--COMMAND ACCEPT character <CTRL-D>

OPT--OPTION character <CTRL-U>

Y/N--ANSWER (y = yes/n = no).

Slashes (/) between prompts indicate alternatives. For example, "C/OK" indicates that the user may either type in a command word or type the COMMAND ACCEPT character.

Other prompts may be generated by parse functions written by grammar writers.

CLI Recognition Modes

The CLI supports several modes for recognizing command words typed by the user. These are described fully in Appendix 9 "User Profile User´s Guide" in the section describing the "Recognition Mode" command.

CLI Character Translation

The CLI allows users to specify that certain characters on her keyboard be used for certain generic functions. For example, the default character for the function COMMAND ACCEPT is <CNTL-D> but a user with no convenient means of typing <CNTL-D> could specify that some other character serve the same function. This character translation feature is described in Appendix 9.

HOW TOOLS INTERFACE TO THE CLI

As discussed in AN INTRODUCTION TO THE FRONTEND, the Frontend uses several interprocess communicaton mechanisms in two modes of communication. Exactly how the tool interfaces to the Frontend depends on these factors.

A tool that uses character oriented communication will obtain a Server Telnet connection by executing a Foreman primitive. (This may not be necessary in some Foreman implementations.) The tool can usually treat this connection exactly as a terminal, and in fact may not be able to tell the difference.

If the tool is run with a transparent grammar it may be written exactly as if the I/O to the Frontend were I/O to a terminal. A more sophisticated grammar may or may not dictate changes in the tool, depending on how the connection is used (determined by the grammar-writer/tool-builder). The grammar may be written enabling it to determine exactly what is sent from the Frontend to the tool, but it is limited to 7-bit characters. Although the information coming from the tool to the Frontend is usually treated as output text to be displayed, the provision is made for the grammar to capture it and deal with it programatically.

In any event, the tool need not concern itself with the communication mechanism, but only with Foreman primitives [3].

The character oriented interface is only intended for pre-NSW tools that do terminal I/O (unsplit tools). Other tools should interface to the Frontend in a message oriented mode.

Tools using message oriented communication (unsplit tools) can make more efficient use of the Frontend. The grammar collects complete commands from the user and invokes procedures in the tool to carry out the commands. Likewise, the tool can invoke procedures in the Frontend to show results or manipulate the display. The communication is characterized by "invoke" and "reply" messages containing structured arguments or result data.

Interprocess Communications Mechanisms

Each communicatons mechanism imposes a data structure and format on the message contents. The current NSW interprocess communication mechanism is MSG-3. Other mechanisms are also described here.

MSG-3. Each MSG-3 to tool interface is host dependent. The message contents are 8-bit bytes following the message format and data structure (NSWB8) defined in Appendices 1 and 3. The tool writer should consult the MSG-3 document for his host.

MSG-1. MSG-1 is an early NSW facility that is no longer supported. A detailed description of its use will not be presented here.

DPS. The Distributed Programming System (DPS) is complete in the sense that it defines the message format and data representation, as well as the tool to DPS interface and DPS primitives available to the tool [4].

Stand Alone (TENEX Subsystem). This mechanism allows the Frontend and the tool to become a single fork TENEX subsystem. The interface is a programming language interface (L10), but does not require that the tool be programmed in any particular language. The interface consists basically of a push-jump (PUSHJ) on register 17 with proper arguments on the stack. The exact calling sequence and data representation is defined in Appendix 2.

Data Representation

In all message oriented communication a structured data representation is assumed. It takes the form of several types of data elements and lists of those data elements or lists. The data structure may be arbitrarily deep. The exact bit configuration is different for different communication mechanisms, but the types are the same:

INTEGER: a twos complement integer.

INDEX: a non-negative number (smaller than integer, allows data compression, enforces a range).

BOOLEAN: TRUE or FALSE value.

BITSTR: a bit string with associated length.

EMPTY: a null data element.

CHARSTR: a character string with associated length.

LIST: a number of these elements with associated length.

See Appendix 3 for detailed definitions of each data representation.

Message Format

Each message communication has the same basic format, a LIST of four elements. Those elements identify the purpose of the message, the transaction identifier (so that replies may indicate which message they are addressing), and a data element to contain the data structure being passed. The exact definition of this list is in Appendix 1.

Tool procedures are invoked from the grammar by making an execution function call in the grammar. The tool procedures are developed by the grammar-writer/tool-builder. Typically, a procedure is designed for each kind of operation the tool performs for the user. Each invocation from the grammar will provide arguments that supply parameters and perhaps modifiers for those operations.

The tool may invoke procedures in the Frontend by sending an "invoke" message to the Frontend. A complete list of available procedures, and their arguments and results, is contained in Appendix 4. These procedures allow the tool to obtain information about the terminal, show results to the user in a terminal independent fashion, and if the terminal is an alphanumeric display terminal, manipulate the display. The display primitives allow the tool to define windows on the display and manipulate text within the windows, independent of the terminal brand.

THE USER PROFILE TOOL

The User Profile tool manipulates a data structure (called the User Profile) that controls the behavior of the system as the user sees it. Through commands in the tool the user may adjust the behaviour of the system to suit his individual preferences. Note that the User Profile influences only tool independent attributes and applies therefore to all tools and subsystems. A complete "User Profile Users´ Guide" is available in Appendix 9.

The User Profile can be divided into two major parts: the first allows the user to tell what tools and subsystems he wants made available to him; the second allows him to control how he wants those tools and subsystems to appear. The first category includes names of programs and tools that he may have made available to him and some instructions on how he wants to start his session with the computer. In the "system appearance" category fall control characters, feedback settings, heralds, prompts, and recognition modes. (Appendix 8 provides a list of the default control characters.)

THE HELP TOOL

Help is implemented as a special tool rather than a Frontend function. When Help is invoked, information about the command being used is passed to the Help tool; it then takes the user to an initial point in the Help description file that describes the command. If the user is not in the middle of specifying a command the tool takes him to appropriate information about the system he is using, or else he may type in a term (or command syntax) and the correct description will be furnished.

A Help description file contains prose definitions of the tool commands, explanations of any special characters of the tool, and a description of how to use the tool. The tool implementers write the file based on the grammar syntax, following a set of conventions listed in Appendix 1Ø.

APPENDIX 1: Frontend Message-Oriented Communication Format

The Frontend message-oriented communication is based on the following message formats, depending on the interprocess communication facility used. The data representation encoding is specified and defined for each in Appendix 3 "Data Representations for Message Communication".

MSG-3 Interface (NSW Transaction Protocol)

The content of MSG messages conform to the following structural outline.

LIST(type, tid, parameter, args)

type: INDEX.

Value of one means invoke an operation.

value of two means that this message is a reply.

value of three means that this message is a response to an alarm.

Other values may be defined as needed.

tid: INDEX.

This is a transaction identifier.

INDEX to indicate acknowledgement required if type=1, or to indicate which invocation request it is acknowledging if type=2.

If type=1 (invoke message):

tid=0 means no acknowledgement required.

tid NOT=0, acknowledgement requested.

If type=2 (acknowledgement message):

tid is the tid of the invocation request.

If type=3 (alarm response):

tid contains the alarm code.

That is, when an operation is invoked the tid specifies whether a acknowledgement is required or not. If required, the same tid must be present in the reply.

For response to an alarm, type=3, this is the alarm code transmitted as an INDEX.

parameter:

if type=1 (invoke request):

CHARSTR which is the name of the operation or procedure to be performed. This string will be interpreted in a upper/lower case independent manner.

if type=2 (acknowledgement) or type=3 (response to alarm):

LIST() (i.e. zero length list), to indicate success, or

LIST(errclass, errnumber, errstring) to indicate an error.

errclass: INDEX.

=1: partial results returned.

This error class is used when several steps are performed by one operation and some of them fail.

=2: failure, resources unavailable.

=3: failure, user error.

=4: failure, NSW error. Recoverable.

=5: failure, NSW error. Fatal.

This indicates that the instance of the NSW component in question is out of business.

errnumber: INDEX.

This is a code for the particular error that ocurred. When possible, these codes should be uniform across different implementations of the same NSW component.

errstring: CHARSTR.

This is a human readable character string describing the error.

args: LIST.

If type=1 (invoke operation)

This is a LIST of arguments.

If type=2 (acknowledgement) or type=3 (response to alarm)

This is a list of results.

The first byte of the message will be the first byte of the above LIST.

Another way to view these format conventions is shown here:

Request Message

```
LIST(

  type: INDEX [ = 1 ],

  transaction-id: INDEX        % zero if no reply is
  required %

  operation-name: CHARSTR,        % operation to be
  performed %

  arguments: LIST(...)
  )
```

Reply Message

```
LIST(

  type: INDEX [ =2 or 3 ],

  transaction-id: INDEX,

  errorcode:  LIST(errclass, errnumber, errstring) or
  LIST()

  results:  LIST(...)

  )
```

MSG Process Names

MSG process names are transmitted as one unit of type BITSTR. The MSG documentation specifies that a process name is composed of several fields (host, incarnation, ...), but this internal structure is not identified at the NSWB8 protocol level.

NSW Procedure Names

In the NSW all externally callable procedures shall be named such that the first two letters of the procedure name identify the NSW component of which that procedure is a part.

The two letter codes are:

FE	Frontend
FM	Foreman
FP	File Package
WM	Works Manager
WO	Works Manager Operator

DPS

The format is determined by DPS. See the DPS-10 VERSION 2.5 PROGRAMMR´S GUIDE [5].

Stand-Alone Frontend (SAFE) Interface

Type of Interface

SAFE interfaces to the Backend through procedure calls. These calls are dispatched through one of two mechanisms: SAFE dispatching or Backend dispatching. The choice of mechanism is up to the Backend writer. (See Appendix 2, "The Stand Alone Frontend (SAFE)".)

SAFE Dispatching

The simpler of the two mechanisms is SAFE dispatching. With it, Backend procedures are called directly with the arguments specified in the grammar and one additional argument. This additional argument is the address of an L10 list that may be used to return results to the Frontend.

Example

CML function call in grammar:

```
fun( cmlvar1, cmlvar2)
```

Corresponding L10 procedure call made by SAFE to Backend:

```
fun( var1, var2, resultlist REF);
```

where var1 and var2 are copies of the cmlvar1 and cmlvar2 and resultlist is empty.

Backend Dispatching

A Backend may need to perform some function each time one of its procedures is called from SAFE, for example, do argument conversion or address resolution. SAFE provides for this by allowing the Backend to do its own procedure calling.

The Backend must supply a dispatcher for this purpose. The dispatcher will be called by SAFE whenever the grammar specifies that a Backend procedure call should be made. Its arguments will be the address of an L10 string containing the name of the procedure to be called, the address of an L10 list of arguments for the procedure, and the address of an L10 list in which to place results. The dispatcher is expected to call the procedure before returning to SAFE.

Example

CML function call in grammar:

```
fun( var1, var2)
```

Corresponding L10 procedure call made by SAFE to Backend:

```
dispatcher( $"fun", $arglist, $resultlist);
```

where arglist contains the values of var1 and var2 and resultlist is empty.

Success/Failure Indication and Returning Results

SAFE expects the Backend procedure it calls to return as its primary result TRUE if the call was successful or FALSE if it was not. This is the case regardless of whether SAFE calls the specific Backend procedure or it calls a Backend dispatcher, that is, regardless of whether SAFE dispatching or Backend dispatching is being used.

If the Backend call was successful, then results (up to 8) put into ´resultlist´ by the Backend will be assigned to the corresponding CML built-in variables RESULT1 through RESULT8. If ´resultlist´ is empty, then RESULT1 through RESULT8 are set to NULL.

If the Backend call was not successful, SAFE expects to find in ´resultlist´ an error number and user-readable string as the first and second elements, respectively. These will be shown to the user by SAFE and then the command will be aborted.

Calling Procedures in SAFE

SAFE provides numerous procedures, including display procedures, which are callable from the Backend. There are two mechanisms for calling these procedures. (See Appendix 2, "The Stand Alone Frontend" for information on how the Backend obtains the addresses of the relevant SAFE procedures.)

The simplest mechanism, analogous to SAFE dispatching, is to make a direct procedure call on the SAFE. Two procedures are accessible in this manner (as well as in ´fprocall´):

```
fshow(

  string REF,  %string to show user%

  boolean  %TRUE means confirmation required%)

  Returns one result

    TRUE if successful call

fshowerror(

  string REF,  %error string%

  boolean  %TRUE means confirmation required%)

  Returns one result

    TRUE if successful call
```

The more powerful mechanism allows abitrary results to be returned by the SAFE procedure. This method requires that the Backend put all the necessary arguments into one L10 list and provide another L10 list for any results that may be generated. The SAFE procedure is called as follows:

```
fproccall(

  string,  %addr of L10 string containing proc name%

  list,  %addr of L10 list of args or 0 if no args%

  list)  %addr of L10 list to get results%

  Returns one result

    TRUE if successful call / FALSE otherwise

      (the result list will contain the results if any)
```

Examples

To tell the user that SAFE is a wonderful thing:

```
fshow( $"SAFE is a wonderful thing", FALSE);
```

To tell the user that the Backend has given up:

```
fshowerror( $"Backend has given up", TRUE);
```

The second arg TRUE indicates that the user must type CONFIRM before continuing. This is appropriate only for critical messages.

To call the SAFE routine that clears a window:

```
fproccall( $"clear-window", $arglist, $resultlist);
```

where arglist is an L10 list containing the arguments for the clear window procedure, i.e. a window identifier, and resultlist is an empty L10 list.

Provision for Termination

SAFE provides a built-in parse function called ´feterminate´. When a grammar calls this parse function, SAFE will process the CML TERMINATION rule for the grammar and then do a HALTF.

Type Conversion

Before passing arguments to an execution function, SAFE converts the CML variable values to appropriate L10 values. In addition, L10 results returned by execution functions are conversely converted. The following table indicates the correspondence between CML values and L10.

CML	L10
string	address of the string
integer	value of the integer
true	TRUE
false	FALSE
list	address of the L10 list
null	value of zero
command word	string if integer token is 0, else list(integer token, string)
point	address of L10 list
address	address of L10 list

Conventions for Languages Other Than L10

The following L10 conventions for the PDP-10 are necessary for anyone NOT using the L10 language when building a stand alone tool.

L10 Calling Sequence

To call procedure "name" with arguments arg1, arg2, ... argn:

```
PUSH S,argn

...

PUSH S,arg2

PUSH S,arg1
```

PUSHJ S,name

[subtract n from both halves of S, or POP n times]

The stack pointer, S, is register 17. Register 12 contains the first result, registers 1, 2, and 3 contain results 2, 3, and 4. Success return is indicated by non-zero register 6, failure by zero in register 6. Registers 16 and 17 MUST be preserved by routine "name".

To call an L10 procedure, follow this sequence. When writing a routine to be called by an L10 procedure, it may be desirable to perform operations such as:

```
name:

  PUSH S,16

  MOVE 16,S

  % now reference argument n as -n-1 indexed by
  register 16 %

  % to return %

  MOVE S,16

  POP S,16

  POPJ S
```

APPENDIX 2: The Stand Alone Frontend

Introduction

The purpose of the Stand Alone Frontend (SAFE) is to provide the L10 programmer (or, with modifications, programmers of other languages) with a powerful, low-overhead user interface that resides in the same fork as the execution (Backend) procedures. This appendix only discusses the L10 version of SAFE; all references to procedure calls, returns, etc. are assumed to obey L10 conventions.

Function of Safe

Execution functions are provided with arguments and called by the Frontend according to instructions it reads in the tool grammar. They may do arbitrary processing and must return a result indicating success or failure. Other results may be returned as well.

SAFE also provides procedures callable by the Backend to show information or error strings to the user, obtain information from him, etc.

Making a Stand Alone Tool (SAT)

Backend Programming Conventions

The Backend must contain a procedure that this document will refer to as ´bestart´. In the process of combining SAFE and the Backend, this routine will be called by SAFE with the addresses of the externally callable SAFE routines:

```
´bestart´( $fproccall, $fshow, $fshowerror)
```

´Bestart´ must return the address of the Backend´s procedure selector array. Here is the format of the procedure selector array:

The first element is a count.

The remainder of the array is a series of count pairs. Each pair consists of the address of a lower case string containing the name of a procedure as used in the grammar and the address of the procedure itself. Here is an example.

```
DECLARE procsel = (3, $"xsearch", $xsearch,  $"xsort",
$xsort,  $"xinsert", $xinsert);
```

IF the Backend writer chooses to do Backend dispatching, the array must have a count of one (1) and a single pair made up of the address of the string "dispatcher" (a fixed name) and the address of the dispatcher:

```
DECLARE procsel = (1, $"dispatcher", $xdispatch);
```

Otherwise, SAFE will do its own dispatching using the Backend provided dispatch table.

Making the Backend Save File

SAFE provides a version of TENLDR, SAFELDR, that has the L10 runtime environment preloaded and the address at which the Backend may start loading (170000B). The programmer runs this version of SAFELDR and starts loading at or above (numerically greater than) 170000B. The programmer should then set the entry vector location to the address of 'bestart'. In a file named BE.SAV he may save whatever portion of the address space between 170000B and 600000B (inclusive) he wishes. (Note that using SAFELDR rather than TENLDR allows the Backend and SAFE to use the same instance of the L10 runtime environment.)

Combining SAFE and the Backend

To combine SAFE and the BE.SAV file previously made using SAFELDR, SAFE must be executed. When SAFE is executed (by typing SAFE.SAV to the EXEC), it will first initialize the L10 environment and itself; then it will load the appropriate grammar (BE.CGR in the connected directory), any parse function data (BE.PFD in the connected directory), and any parse function code (BE.PFC in the connected directory). All three of these will be loaded below SAFE-END, i.e., inside of SAFE.

Next, SAFE will GET BE.SAV (from the connected directory into its own address space) and do an L10 procedure call to the entry vector location, i.e., to 'bestart'. 'Bestart' will be called by SAFE and is expected to follow the conventions described above.

Upon return from 'bestart', SAFE will do whatever processing it desires, then set its entry vector location and do a HALTF. The Backend programmer should then save the entire address space on whatever file is desired. The entry vector will be set so that when the file is started, execution will begin with SAFE processing the CML INITIALIZATION rule for the tool.

APPENDIX 3: Frontend Data Representations for Message Communication

NSWB8 Data Representation (previously PCPB8)

Data Structure Types and Encoding

This is an eight (8) bit byte encoding of a set of basic data types for use in computer communications protocols. The first byte of a data structure is a type code; the following bytes depend on the type code. Type code zero is reserved. Type code 8 is reserved for possible REPEAT (compression) use in the future. Type code 10 is reserved for use of EBCDIC strings, although such strings are not currently transmitted between hosts.

EMPTY

TYPE (1 byte) = 1

VALUE (none) empty

BOOLEAN

TYPE (1 byte) = 2

VALUE (1 byte) boolean

FALSE = 0

TRUE = 1

INDEX

TYPE (1 byte) = 3

VALUE (2 bytes) index

The value represents a positive integer in the range 0 through 2**15 - 1 (** denotes exponentiation).

The most significant byte is first.

INTEGER

TYPE (1 byte) = 4

VALUE (4 bytes) two´s complement integer

The most significant byte is first.

BITSTR

TYPE (1 byte) = 5

COUNT (2 bytes)

VALUE (count bits) left adjusted in ((count+7)/8) bytes)

CHARSTR

TYPE (1 byte) = 6

COUNT (2 bytes)

VALUE (count bytes) ascii text

LIST

TYPE (1 byte) = 7

COUNT (2 bytes)

Count Data Structures

PAD

TYPE (1 byte) = 9

VALUE (none)

Any PAD elements should be completely ignored. They are not to be counted (for example, as elements of a LIST). The concept of a PAD element has been useful in forming data structures, especially when the structure cannot be built sequentially.

Data Structure Format

element

```
          *------*
```

```
empty    *  1  *
         *-----*
            1

         *-----*--------*
boolean  *  2  * Ø or 1 * Ø for FALSE or 1 for TRUE
         *-----*--------*
            1      1

         *-----*-------*
index    *  3  * index * small positive integer
         *-----*-------*
            1      2

         *-----*---------*
integer  *  4  * integer * two´s complement integer
         *-----*---------*
            1       4

         *-----*-------*------*
bitstr   *  5  * count * bits *
         *-----*-------*------*
            1      2     count   ((count+7)/8 bytes)

         *-----*-------*------*
charstr  *  6  * count * text * Network ASCII
         *-----*-------*------*
            1      2     count

         *-----*-------*------------------*
list     *  7  * count * count-structures *
         *-----*-------*------------------*
            1      2
```

Examples

Empty

```
*-----*
*  1  *
*-----*
```

Boolean "TRUE"

```
*-----*-----*
*  2  *  1  *
*-----*-----*
```

Index "7"

```
*-----*-----*-----*
*  3  *  0  *  7  *
*-----*-----*-----*
```

Integer "-3"

```
*-----*-----*-----*-----*-----*
*  4  * 255 * 255 * 255 * 253 *
*-----*-----*-----*-----*-----*
```

Bit string "1000111110I011"

```
*-----*-----*------*-----*-----*
*  5  *  0  *  14  * 143 * 172 *
*-----*-----*------*-----*-----*
```

Character string "ABCDE"

```
*-----*-----*-----*-----*-----*-----*-----*-----*
*  6  *  0  *  5  *  A  *  B  *  C  *  D  *  E  *
*-----*-----*-----*-----*-----*-----*-----*-----*
```

List of a character string "ABC" and a Boolean "FALSE"

```
*-----*-----*-----*-----*-----*----
*  7  *  0  *  2  *  6  *  0  *  3
*-----*-----*-----*-----*-----*----

  -*-----*-----*-----*-----*-----*
   *  A  *  B  *  C  *  2  *  0  *
  -*-----*-----*-----*-----*-----*
```

PCPB36 Data Representation

Data Structure Encoding and Format

The data structure consists of 36 bit words. The first word or two of each data element contains fields that determine the element type and in some cases, length and/or value.

bits 0-13: unused (zero)

bits 14-17: type

bits 18-20: unused (zero)

bits 21-35: value or COUNT

EMPTY

type = 1

value unused (zero)

element size: 1 word.

BOOLEAN

type = 2

value (bits 21-34 zero)

FALSE=0 bit 35

TRUE =1 bit 35

element size: 1 word

INDEX

type = 3

value index in bits 21-35

The value represents a positive integer in the range 1 through 2**15 - 1

element size: 1 word

INTEGER

type = 4

value integer in next word, 36 bits

element size: 2 words

BITSTR

type = 5

COUNT in bits 21-35

value (count bits) left adjusted in ((count+35)/36) words) element size: 1+((count+35)/36) words

CHARSTR

type = 6

COUNT in bits 21-35

value (count 7-bit bytes) ascii text

in next ((count+4)/5) words, 5 bytes per word

element size: 1+((count+4)/5) words

LIST

type = 7

COUNT in bits 21-35

Count Data Structures in following words

element size: 1 + count data structures

PAD

type = 9

value unused (zero)

Any PAD elements should be completely ignored. They are not to be counted (for example, as elements of a LIST). The concept of a PAD element has been useful to us in forming data structures, especially when the structure cannot be built sequentially. However, we do not expect the FE to send messages with PADs in them.

L10 Data Representation

Data Structure Encoding and Format

L10 data representation is comparable to PCPB36 but is an in-memory data structure with memory addresses in it, and it is not sequential. Typically, the address of a data structure is passed to a procedure, and that data structure is an L10 LIST. L10 LIST format is described below. Each word in the LIST represents a data element, with fields used as follows:

bits 0-7: maintained by L10 runtime, zero if storage maintained by program.

bit 7: maintained by L10 runtime, zero if storage maintained by program.

bit 8: immediate.

bits 9-17: type.

bits 18-35: value.

EMPTY

- type = 0
- IMMEDIATE = 0
- value = 0

INDEX

- type = 9
- IMMEDIATE = 1
- value = index in bits 18-35.

INTEGER

- type = 1
- IMMEDIATE = 0
- value = 36 bit word at address in bits 18-35.

BOOLEAN

- type = 17
- IMMEDIATE = 1
- value in bits 18-35
  - TRUE = one
  - FALSE = zero

CHARSTR

type = 2

IMMEDIATE = Ø

value = L1Ø string at address in bits 18-35. (See below.)

BITSTR

type = 4

IMMEDIATE = Ø

value = L1Ø bitstr at address in bits 18-35. (See below.)

LIST

type = 3

IMMEDIATE = Ø

value = L1Ø LIST at address in bits 18-35. (See below.)

L1Ø Data Element Formats

L1Ø LIST

indirect (normally zero, see below).

list: max,,len ;current and max number of elements.

[len words, one element per word formatted as above].

[A list of length 1 contains one element at address list+1].

Note:

´indirect´ is the word at address list-1. Non-zero here indicates that the list elements do not reside at list+1 etc., but at indirect+1 etc. ´indirect´ is actually the address of another L1Ø LIST. The length at list must be the same as the length at indirect.

L1Ø STRING

string: max,,len ;current and max number of characters.

[(len+4)/5 words with containing len characters, 5 per word, 7 bits per character].

[A string of length 1 contains one character, the most significant 7 bits at address string+1].

L10 BITSTR

```
bitstr: len   ;bit count in one full word.
```

[(len+35)/36 words with bits left justified].

[A bitstr of lenth 1 contains one bit, the most significant bit at address bitstr+1].

APPENDIX 4: Externally Callable Procedures in the Frontend

Introduction

This is a description of the externally callable procedures in the Frontend. Each description includes the argument and result data structures, and the function to be performed. These procedures allow the tool to obtain information from the Frontend, display information to the user, and otherwise manipulate the user's terminal.

While procedure names are written in lower case letters in the appendix, the actual character string containing the procedure name in the invoke message may be in upper or lower case characters.

Notation

The procedure name, arguments, and results are presented as

```
<procedure-name> ( <arguments> -> <results> )
```

where the arguments and results are structured data elements. Elements enclosed in square brackets [] are optional. The exact data representation depends on the communication medium being used--either PCPB36, PCPB8, or L10 structure.

The data elements for each procedure are then given a symbolic name and defined; the data type is also provided:

```
<element-name>:  <type>

  <explanation>.
```

The data types are INTEGER, INDEX, BOOLEAN, CHARSTR, BITSTR, EMPTY, and LIST. CHARSTR, BITSTR, and LIST have lengths asociated with them. LIST elements may each be any of these data elements.

Packages

The procedures are grouped into "packages" or categories as follows:

tool-package

show, show-error, help.

wm-package

new-node-profile, new-profile, show, show-error, help.

up-package

new-profile.

dpy-package

get-windows, batch-display-commands, create-window, delete-window, clear-window, scroll-window, set-window-attribute, write-string, write-line-segment, replace-string, delete-string, reposition-string, set-string-attribute, reposition-line-segment, delete-line-segment, set-line-segment-attributes, write-literal.

Presenting Information to the User (Normally Done Through CML Statements)

show (message, confirmflag [, format])

The message is presented to the user. If confirmflag is TRUE, the user may not continue until he confirms that he saw the message. In this case, the SHOW procedure does not return to the caller until the user has confirmed the message. On display terminals, the message will appear in the command feedback window.

message: CHARSTR

This may contain formatting characters such as carriage return <CR> or linefeed <LF>.

confirmflag: BOOLEAN

If TRUE, the user must confirm that he saw the message before he may continue.

format: INDEX

This optional argument provides format control as follows:

=1: <CR>, <LF> after message.

=2: <CR>, <LF> before message.

=3: <CR>, <LF> before and after message.

show-status (message, confirmflag)

This procedure is used to present error or warning messages to the user. Confirmflag is used as in SHOW. On display terminals, the message will be displayed in the TTY window.

Getting Help from the User

help (helptype, abortmessage, helpmessage [, arg1, ... arg8] -> helpresult)

HELP is called by an execution function that needs information from a user. (For example, the execution function may have received a mistyped argument from the user.) The helpmessage is typed to the user. Depending on helptype, the information goes through several mechanisms. If the user aborts the help by typing COMMAND DELETE, the abort message will be presented to the user. If help is obtained, it will be returned as the result of the call. See Appendix 7 for a discussion of HELP rules.

helptype = 1 is for bad argument help.

The number of the argument is arg1, which in this case must be an INTEGER. If the third argument received by the execution function is bad, then arg1 should be 3. Arg2 through arg8 are not used. The CLI determines where the argument was collected and collects a replacement, returning it as helpresult.

helptype = 2 is for executive grammar help.

The executive grammar (the one the user interacts with when the CLI starts) may have associated with it another grammar, called a help grammar. In this case, the value of arg1, an INTEGER, is given to the built-in CML variable HELPCODE. The help grammar determines what kind of help to get from the user based on HELPCODE and asks the user for it. The HELP obtained, if any, is returned as helpresult.

helptype = 3 is for tool grammar help.

Each tool grammar may have in it a single rule to be used as a HELP rule anywhere within the grammar. Arg1 is used to set up HELPCODE, as is done with executive grammar help, and the HELP rule may use it arbitrarily. The HELP rule must do a RESUME(var) if it obtained help and the value of var will be returned as helpresult.

helptype > 100 is for execution function-call help.

Each time an execution function is called in a grammar, the grammar writer may specify that a specific rule is to be invoked if HELP is called with helptype greater than 100. In this case, HELPCODE is given the value of helptype and arg1 through arg8, if present, are put into CML built-ins RESULT1 through RESULT8. The HELP rule should do a RESUME, as in the tool grammar help above, and the value of the variable will be the result of the call.

helptype: INDEX

abortmessage, helpmessage: CHARSTR or EMPTY

arg1-arg8: depends on helptype

helpresult: any type

Getting Terminal Characteristics

get-windows (-> characteristics)

This procedure allows the tool to find out general information about the terminal class and configuration. The Frontend knows the terminal brand; the tool need only concern itself with the general capabilities of the terminal.

characteristics: LIST (terminal-class, default-text-window, primary-window, mode-window, other-windows)

terminal-class: INDEX

If the terminal-class is less than 3, the display manipulation procedures may not be used.

=1: line-at-a-time and/or half duplex terminal.

=2: full duplex typewriter terminal.

=3: full duplex display terminal with or without pointing device.

default-text-window: window-info

This is the simulation-TTY window, which receives all show-error messages, linking text, and local host terminal output (if any).

primary-window: window-info

At tool startup time, this is the only existing window for that tool. All windows that tool wishes to create must reside within this window. The tool cannot delete this window.

mode-window: window-info

The mode-window is a small, sequential window of two lines, with eight characters each. window the tool may write in to . It is used to show the user information, most typically various tool operating modes. The tool may write in the window and clear it, but may not manipulate it in any other way. There is one such window for all tools.

other-windows: window-info, window-info...

These are windows that have been created by the tool. If terminal-class is less than 3, or the tool has not created any windows, these will not be present.

window-info: LIST (owning-window-id, window-id, type, diag-coords, window-att)

With respect to the coordinate system of the owning window, diag-coords are upper-left and lower-right. In the case of the graphics and default text windows, the coordinates are with respect to the virtual coordinate system of the terminal (or combination of physical terminals). In any type of window, the coordinates (Ø,Ø) represent the lowest left-most corner of the window.

See definitions below under "create-window".

owning-window-id: window-id or EMPTY

An owning-window-id of EMPTY means that there is no owning window. That is the case for printer and graphics windows, and for the full-screen window used during teletype simulation.

window-id: INDEX

This index is a designator for the display window in question.

Manipulating Display Terminals

Introductory Notes

Class three terminals (full duplex displays) are either Data Media Elite 2500 alphanumeric terminals (expansion to other brands on request) or Line Processor terminals [6, 7]. Line Processor terminals have a pointing device (mouse) and may have a Tektronix storage tube device attached in parallel to the alphanumeric display. The "graphics" operations described here currently function only on Line Processor terminals with Tektronix storage terminals attached.

The Frontend will allow certain of the display primitives to be used for graphics display manipulation. This is detected on the basis of the window-id used in the call. Output to printers that are part of the terminal configuration (used in conjuction with Line Processors) is also detected on the basis of the window-id. For the graphics terminal, the Frontend can mark selections made by the user on the graphics display by drawing a dot at the specified location. Any other marking must be done by the tool using the write-literal procedure.

Display screens are divided into windows, which are rectangular screen areas. There are several types of windows, the most common of which are random and sequential. In random windows text can be written, moved, deleted, or changed. Sequential windows simulate the operation of a typewriter. In sequential windows text can only be appended.

Random windows contain strings--collections of line segments. Each line segment is an unbroken text string with its own origin (i.e., character position on the screen, identified by horizontal and vertical screen coordinates). Thus, line segments within a string need not be adjacent, nor must they be grouped visually (although they usually are). If a line segment will not fit in the specified area, it will be truncated at the right margin of the (random) window.

The Frontend keeps a data base describing the window contents. When a tool manipulates the display, it is manipulating that data base through procedure calls. Elements of the data base are referenced by window, string, or line segment identifiers defined by the Frontend and returned to the tool when the window, string, or line segment is created. It is the responsibility of the Frontend to see that the display screen corresponds to the data base.

Each type of element (window, string, line segment) has attribute bits associated with it that determine how it will be displayed. These bits control the following display attributes: visible (1)/invisible (0); normal-lighting (1)/highlighting (0); and non-printing-invisible (1)/non-printing-visible (0). In each case a zero represents a normal mode and a one represents an exceptional mode. When a line segment is written, its attribute bits and those for its window and string are combined with a logical OR operation. Hence to make an entire window display in an exceptional mode, the tool turns that attribute on in the window only, not each line segment. This allows any line segments or strings that are displayed in an exceptional way to remain as they are when the window is returned to normal.

Two additional attributes apply to windows only. The attribute autostop applied to a sequential window causes the Frontend to wait for the user to type a confirmation character before contents of the window scroll out of view, in a manner similar to the TENEX terminal type scope command. The attribute wordbreak applied to a sequential window causes lines to be broken on word boundaries when the text between carriage returns will not fit on a single line.

Many Changes to the Screen at Once

batch-display-commands (display-commands -> ids);

This routine is used to effect many changes to the screen at once. The display-commands argument is an encoded data structure that results in calls on the display primitives described below.

display-commands: LIST(LIST (opcode, params), ...)

Any number of pairs of procedure opcodes and parameters may be specified. The procedures will be called in the order they appear in the above list.

Keep in mind that the tool is manipulating the display data base. The operations must be performed in meaningful order. For example, writing a line segment and then scrolling is not the same as scrolling and then writing a line segment, if the line segment is in the scrolled area.

For best results, put as many operations into a single batch-commands call as possible. Always put any clear-window calls as the first operation(s) in the batch-commands list.

opcode:

INDEX (designates procedure--see table below)

EMPTY (default to last procedure index used)

params: as appropriate for given procedure

procedure names:

1: get-windows

2: batch-display-commands (batch nesting not recommended)

3: create-window

4: delete-window

5: clear-window

6: scroll-window

7: set-window-attributes

8: write-string

9: write-line-segment

10: replace-string

11: delete-string

12: reposition-string

13: set-string-attributes

14: reposition-line-segment

15: replace-line-segment

16: delete-line-segment

17: set-line-segment-attributes

18: write-literal

19: track

ids: LIST (results)

The batch-commands results are exactly the results of each procedure incorporated into a list. Procedures that produce no results do not produce an element in the batch-commands result list.

Window Manipulation

create-window (old-window-id, type, diag-coords, new-window-att -> window-id)

This procedure creates a new window with respect to an old window. The coordinate system is expressed in terms of coordinate units relative to the old window. For alphanumeric display screens, coordinate units correspond one to one to character positions.

new-window-att: EMPTY or window-att

If EMPTY, old window values are used.

window-att: LIST (window-priority, string-att)

Window priority and attributes control how the text in the window appears to the user.

The coordinate pair in string-att is ignored.

window-priority: INDEX

Window priority is an integer from 1 to 7 (1 having the highest priority). Whenever two windows overlap, the text of the higher priority window will dominate. (Note that this may only effect the overlapped area, not necessarily the whole window.) The CLI´s command feedback window is always higher priority than tool windows. The window that is created by the CLI when it starts a new tool will have priority 1 for that tool. The tool may change this if desired.

old-window-id, window-id: INDEX

diag-coords: LIST (upper-left-x, upper-left-y, lower-right-x, lower-right-y)

upper-left-x, upper-left-y, lower-right-x, lower-right-y: INTEGER or EMPTY

These are coordinates defining window size and location in terms of old-window relative coordinates. All coordinates must be positive. EMPTY defaults to the corresponding coordinate for the old-window.

type: INDEX

1= random (allows arbitrary text manipulation)

2= sequential (use write-literal, clear, delete wndow)

3= graphics (arbitrary writes, clear, delete window)

4= printer (use write-literal, delete window)

There are limitations, as follows:

The new window must lie within the old window, unless it is a graphics or printer window.

Create-window for printer or graphics windows will FAIL if hardware is in use already. Only one tool may have graphics windows or the printer window at a time.

delete-window (window-id)

This deletes the specified window. The window-id will no longer be valid, nor, of course, will any string-ids or line-segment-ids that belonged in that window. Any image of the window on the user´s terminal is cleared. If a window "owns" other windows, they are deleted also.

Tools can only delete windows that they created. Windows created for the tool at run-tool time (primary window) do not belong to the process and thus cannot be deleted. They can be manipulated in other ways, however. The default-tty window cannot be deleted.

clear-window (window-id)

This procedure deletes the contents of the window, frees all string-ids and line-seg-ids, and removes the image from the user's display. It may be used for graphics or printer windows.

scroll-window (window-id, change, ycord-top, ycord-bottom)

The two lines indicated define a sub-window within the window specified by window-id. This command affects only lines within that sub-window. Those lines are scrolled "change" lines. Line segments that would fall out of the sub-window after the scroll are deleted. If all line segments of a string are deleted, the string will be deleted also.

Vacated lines are blank after the scroll. For example, if "change" were +2 there would be two blank lines at the bottom of the sub-window when the command was completed. A sub-window may be cleared by giving "change" the value of plus or minus the number of lines in the sub-window.

change: INTEGER

A positive integer indicates scroll UP change lines, a negative integer indicates scroll DOWN change lines.

ycord-top, ycord-bottom: INTEGER

These top and bottom window-relative y coordinates define a sub-window to be scrolled. Line ycord-top through and including ycord-bottom are changed; the rest of the window is unchanged. EMPTY ycord-top or EMPTY ycord-bottom default to the window top line or bottom line respectively.

set-window-attributes (window-id, type, diag-coords, window-att)

This sets the specified attributes for the window. If the size is changed, the window must still reside within the owning window. The type may only be changed from random to sequential or sequential to random, in which case a clear-window is implied. The diag-coords for the primary window may not be changed.

String Manipulation (Random and Graphics Windows Only)

write-string (window-id, string-att, string -> string-id, LIST(line-segment-id, ...))

This writes the specified string at the specified location with the specified attributes. The string consists of individually attributed and positioned line segments (which do not cross line boundaries). Identifiers for the string as a whole and for the individual line segments are returned.

string-att: line-seg-att or EMPTY

If EMPTY, write-string will use window values as string defaults.

This MUST supply coordinates for the origin of the string.

string: LIST(linesegment, ...) or string-addr

This designates the new string to be written (or copied).

linesegment: LIST (line-seg-att, linesegment-text)

Each line segment may have attributes specified for it. The coordinates here are relative to the window origin.

linsegment-text: CHARSTR or line-seg-addr

The acutal text may be given, or existing text may be copied.

line-seg-att: EMPTY or LIST (cords, att-bits, selcode [, char-size, horiz-space])

Attributes include position, view mode (att-bits), and selector code, and are associates with every window, string, and line segment.

If EMPTY the proper defaults are applied.

The optional char-size and horiz-space are REQUIRED for windows of type graphics, and must NOT be present for window type random. They determine the character size and horizontal inter-character spacing for the string.

char-size: INDEX or EMPTY

The graphics character size may be an integer of value one through four. The standard horizontal spacing for size (1, 2, 3, 4) is (14, 13, 9, 8).

horiz-space: INDEX or EMPTY

The horizontal spacing is in device raster units.

selcode: INDEX or EMPTY

Every line segment has a selector code associated with it. The code determines when and how that line segment may be selected by pointing to the screen. This allows line segments to be selectable in some contexts and not in others.

Low numbered selector codes are defined as follows:

=1: The line segment is only selectable for a literal selection (LSEL), not for source or destination selection (SSEL or DSEL). This implies that the text is not part of the tool-controlled data base, but the text may by pointed to in lieu of typing it in.

=2: The line segment may be selected for source and destination (SSEL or DSEL) but NOT for literal selections (LSEL). This implies that the text on the screen is in the tool-controlled data base but that the tool does not want the Frontend to copy characters, text, or words directly from the Frontend´s display data base.

This is the case if an editor writes "<CTRL-L>" instead of a real <CTRL-L>. If the user made a selection of type LSEL on one of the characters of that string and the selcode were 1, the user would get "<" for example, not a <CTRL-L>. If the selcode were 2, the LSEL would not be resolved--the tool would get an address (window, string, linsegment ids) and the tool could resolve it to a <CTRL-L>.

In the case where the user is making a cross-tool literal selection, the literal is resolved by the Frontend in spite of the selcode.

=3: This is the normal case for tool-controlled text. All kinds of selections are allowed.

> 3: The grammar may specify an exceptional kind of selector code (greater than 3) and only line segments with matching selector code may be selected. The current upper bound for the selector code is 255.

att-bits: INDEX or EMPTY

This is a positive integer with bits meaning as follows:

bit 0 (rightmost bit) = 1 (ALWAYS)

bit 1 (highlight)

highlight TRUE: Make this line segment stand out from the rest of the text on the display (in a manner that is appropriate for the device). This may make the text bright or blinking.

highlight FALSE: display normally.

bit 2 (invisible)

invisible TRUE: do not display this line segment, string, or window.

invisible FALSE: display normally.

bit 3 (non-printing-visible)

non-printing-visible TRUE: Any non-printing characters in the line segment, string, or window will be displayed as a highlight printing character. Thus characters that do not normally have a visible representation on the display can be made visible. A single visible character will be displayed in place of the non-printing character. Currently <TAB>, <CR>, <LF>, <RUB> and <SP> are displayed as >, *, vertical bar, #, and tilda. The other non-printing characters are displayed as a capital letter, such as A for <CTRL-A>.

bit 4 (autostop)

has meaning only for sequential windows. If TRUE, the Frontend will stop appending to the sequential window when information in it may be scrolled out of the user´s view; otherwise this will not take place.

bit 5 (word-break)

has meaning only for sequential windows. If TRUE, when appended text reaches the right margin of the window before a carriage return is encountered, the line will be broken on a word boundary if possible. Otherwise, every position on the line will be filled and the remaining text will be written on the next line.

cords: LIST (xcord, ycord) or EMPTY

xcord, ycord: INTEGER or EMPTY

For alphanumeric display, coordinates are in character, line positions. For graphics display coordiantes are in hardware raster units.

Coordinates are relative to window origin. Hence 0,0 is the lower left corner of a window.

EMPTY defaults to appropriate values.

string-addr: string-id / LIST(window-id, string-id)

This designates either a string in the same window or another window to be copied.

string-id: lin-segment-id: INDEX

line-seg-addr: LIST (window-id, string-id, line-seg-id) / LIST (string-id, line-seg-id)

This designates an existing line segment to be copied.

replace-string (window-id, string-id, string-att, string -> LIST(line-segment-id, ...))

This replaces the specified string with a new string or with a copy of a string already in a window belongiing to this process. Note that the old string-id now applies to the new string. Note also that the string´s position within the window can be changed during the replace.

All line-segment-ids in the old string are freed and a new list of line-segment-ids is returned.

Any EMPTY values default to the string-id values.

move-string (window-id, string-att, string-addr -> string-id, LIST (line-segment-id, ...))

NOT IMPLEMENTED AT THIS TIME.

Set-string-attributes can be used to change the position of a string within a window. Move-string can be used to move a string from one window to another. It is equivalent to using write-string to copy a string and then delete-string to delete the old copy.

delete-string (window-id, string-id)

This deletes the specified string and frees the string-id. All line-segments that are part of the string are deleted also, of course.

set-string-attribute (window-id, string-id, string-att)

This sets the specified attributes for the specified string. Note that the position of the string within the window can be changed with this primitive. Default values will be taken from the old string.

reposition-string (window-id, string-id, cords)

This is just a special case of set-string-attributes, but the frequency with which it is done warrents a separate, more efficient call.

Line Segment Manipulation (Only in Random and Graphics Windows)

write-line-segement (window-id, string-id, linsegment -> line-segment-id)

This append a new line segment to the specified string. The identifier for the new line segment is returned. Default values will be taken from the string. Note that this serves the copy function also.

replace-line-segment (window-id, string-id, line-seg-id, linesegment)

This is used to replace a specified line segment with a new or a copy of an old line segment. Note that attributes of the line segment can be changed during the replace.

Default values will be taken from the old line segment. The line-segment-id remains the same.

delete-line-segment (window-id, string-id, line-segment-id)

This deletes the specified line segment and frees the line segment id.

move-line-segment (window-id, string-id, linesegment -> lin-seg-id)

NOT IMPLEMENTED AT THIS TIME.

This is equivalent to using write-line-segment to copy a line segment and then delete-line-segment to delete the old copy.

set-line-segment-attributes (window-id, string-id, line-seg-id, line-seg-att)

This sets the specified attributes for the specified line segment.

reposition-line-segment (window-id, string-id, line-seg-id, cords)

This is just a special case of set-line-segment-attributes, but its frequency warrents a special, more efficient call.

Secondary Device Manipulation

write-literal (window-id, literal-string)

literal-string: CHARSTR

This is treated in a window dependent manner. The type of window-id will tell the FE how to get the literal string to the correct place.

graphics window:

This primitive is used to mainpulate the graphics device (other than to write strings and clear the window). The literal string will be passed to the device unchanged.

printer window:

The literal string will be scanned for TENEX line printer control characters and a "best effort" will be made to produce a hardcopy, formatted as though sent to a line printer.

sequential window:

The literal string will be written into the sequential window. <CR> and <LF> will cause a simulated carriage return and line feed in the window. No padding is necessary.

A sequential window can be scrolled by sending line feeds into it via this primitive, or by using the scroll-window primitive.

track (switch)

switch: BOOLEAN

FALSE: Causes the Frontend to stop tracking the pointing device associated with the graphics device.

TRUE: Causes the Frontend to resume tracking the pointing device.

This procedure is used by the tool to control tracking in the graphics windows, to allow the tool to position the graphics cursor (via write-literal) and assure that the Frontend will not change the cursor position between procedure calls to manipulate the graphics windows.

The tool must be sure to call track(TRUE) when it is finished manipulating the graphics device, or the user will not be able to point to graphic entities.

Getting Direct Connections

Introductory Notes

Under some conditions it is possible and desirable to get direct network connections to the Frontend. These procedures control opening and closing of these connections. Currently, they may only be called if the communication facility is MSG-3 and they are used to establish a Telnet connection for an unsplit tool, or an 8-bit binary send/receive connection for a split tool.

feopenconn (type, processname, connid ->)

This may be used to open a connection of kind and purpose designated by type, to process processname. An acknowledgement is not required. The Frontend will obtain the specified connectin to the specified process from MSG-3.

type: INDEX

This determines the type and semantics of the connection as follows:

=1: Telnet connection for typewriter-oriented tools.

=2: 8-bit binary send/receive connection pair for split tool message-oriented communication. This is an interim communication mechanism to be used until MSG-3 process introduction is implemented.

Other values may be defined as needed.

processname: BITSTR or EMPTY

This is the MSG-3 process name to which the connection will be directed. The EMPTY defaults to the caller process name.

connid: INDEX

This is the connection identifier that will be used in the OPENCONN primitive used by the Frontend, to establish the connection. This identifier is also used to when the connection is closed.

fecloseterm (termtool, processname, connid ->)

This procedure is called to close the specified connection and possibly terminate the tool.

termtool: BOOLEAN

If TRUE, termtool closes the connection AND presumes that the tool has terminated. If FALSE, it simply closes the connection. For connections obtained by calling feopenconn with type equal to 1 or 2, it is expected that termtool will be TRUE.

processname: same as for feopenconn.

connid: same as for feopenconn.

fedonetool (processname, [showstring] ->)

This procedure is called by the Works Manager to inform the Frontend that the indicated tool is no longer being used and has been stopped.

processname: BITSTR

This is the process name for the Foreman controlling the tool in question.

showstring: CHARSTR

This optional item is simply a string that is shown to the user. It normally contains computer charge information.

Profile Updating Procedures

new-profile (user-id, profile)

NOT YET IMPLEMENTED

This allows the USER OPTIONS tool to update the user´s interaction profile in the FE when the user runs this tool to adapt the FE interface. The profile is assumed to be a simple bitstr that is already properly set up for use by the FE. The exact meaning of the bitstr is defined elsewhere.

user-id: INTEGER

profile: BITSTR

new-toollist (user-id, toollist)

NOT YET IMPLEMENTED

This allows the WM to update the list of tools this user is allowed to run whenever conditions warrent. This list is used to give help and tool name recognition to the user. It in no way grants the user actual access to the tool.

toollist: LIST (toolnamelist, entrytool)

entrytool: INTEGER or EMPTY

The integer is an index into tool list.

toolnamelist: LIST (utoolname, wmtoolname, ...)

This list contains associates user´s tool names and Works Manager´s tool names. Any number of pairs may be present.

utoolname: CHARSTR

wmtoolname: CHARSTR

APPENDIX 5: Parse Functions

Introduction

CML provides facilities that satisfy most user interface system requirements. Those requirements that remain unsatisfied are typically not of general utility. For example, a graphics system may need to collect a coordinate from the user and immediately check that it falls within a certain interval. CML provides for the implementation of such special purpose operations by allowing functions written in a high-level programming language to be called from arbitrary points within the grammar. These functions are referred to as parse functions (PFs) because they may affect the parsing of user input by the grammar.

The following discussion assumes that the reader is familiar with the CML and L10 languages in general and L10 coroutines in particular.

Parse Function Description and Operation

Overview

Parse functions are written as L10 coroutines that conform to certain conventions imposed by the CLI. They may do arbitrary processing, including reading characters from the user. They must determine whether the CLI is to continue processing the grammar past the point where the PF was invoked.

Arguments

Parse functions may be invoked in a CML grammar with from 0 to 8 arguments. The CLI puts pointers to these CML arguments into a block. When the CLI OPENPORTs the PF, a pointer to this block is stored as the fifth OPENPORT argument (called ´arguments´ in the model below). The following illustrates the first statement of a PF OPENPORTed by the CLI, and a description of all the OPENPORT arguments for any parse function.

```
(parsefun) COROUTINE ( reason, instruction, accumulator
REF, argcount, arguments REF, saveword );
```

´reason´. The reason the PF is being invoked. It may assume any of the following values, which are declared as external constants in the Frontend.

'parsing'--the CLI has reached a point in the grammar where this PF should be invoked. It will be PCALLed again to do its processing.

'terminate'--the CLI had previously invoked this PF with reason 'parsing', the PF has completed, and the end of the command has now been reached. The PF is expected to do any cleanup necessary, such as freeing space it had allocated. It must do this cleanup in its PORT ENTRY EXIT block because it will not be PCALLed again.

'abortcmd'--the CLI had previously invoked this PF with reason 'parsing', the PF has completed, and the command was subsequently aborted. The parse function is expected to clean up to restore the context that existed previous to this command (e.g., by resetting global variables to their previous value and freeing allocated space.) It must do this in its PORT ENTRY EXIT block.

'backup'--the CLI had previously invoked this PF with reason 'parsing' and the user subsequently backed up the command. The PF is expected to take action similar to 'abortcmd'. It must do this in its PORT ENTRY EXIT block.

'instruction'. Byte pointer to the first byte of this CLI instruction in the grammar.

'accumulator'. Address of the CLI accumulator.

'argcount'. The number of CML arguments the PF was called with, listed in the grammar.

'arguments'. Address of a block containing pointers to the CML arguments for the PF, as listed in the grammar. Take, for example, a parse function referenced in the grammar as:

```
parsef( snip, snap, snur)
```

It will be called OPENPORTed, with the six arguments listed here. 'argcount' will have value 3 (the three CML arguments) and 'arguments' will point to the block containing pointers to the values of CML variables snip, snap, and snur, respectively.

'saveword'. When the PF is invoked with reason 'parsing', 'saveword' is 0. When the PF is invoked for any other reason, 'saveword' is the value the PF asked to have saved when it was originally invoked with reason ´parsing´. This is explained fully in the "Operation" section, below.

Results

Parse functions may produce a result which is accessible to the grammar, e.g., by assigning the result to a CML variable. The result must be a valid CML variable value; it is returned by putting a pointer to it in the accumulator argument.

Suppose a parse function ´numberpf´ was to produce some number which would subsequently be shown to the user. The CML might be

```
cmlvar _ numberpf() SHOW( cmlvar )
```

or more concisely,

```
SHOW( numberpf() )
```

The PF could be as follows:

```
(numberpf) COROUTINE ( reason, instruction, accumulator
REF, argcount, arguments REF, saveword REF );

  LOCAL number, cmlvalue REF;

  ...

  number _ ackerman( 10 );  % generate a number to show to
  user %

  &cmlvalue _ getblk( 3, $freespace); % get a block from
  freespace to put CML value into %

  cmlvalue.vtype _ integer; % set up type of CML value %

  cmlvalue.vlength _ 2; % and length %

  cmlvalue[1]   number; % put the number into the CML value
  %

  accumulator _ &cmlvalue; % put a pointer to the value
  into accumulator %

  ...

  END.
```

Operation

A PF often interacts with the CLI during the specification of a command in which it is invoked. It is first OPENPORTed with reason ´parsing´. It may then get characters from the user through the CLI, ask the CLI to resolve whether or not it is on the correct path through the grammar, and tell the CLI that it has finished processing. When the command is completed or aborted, it may be OPENPORTed again to do some cleanup, including freeing storage it has allocated.

After a PF has been OPENPORTed with reason ´parsing´, it interacts with the CLI by PCALLing to its caller, with the first argument indicating the reason for the call, and by being PCALLed back. The possible values for the first PCALL argument are:

´nextchar´. The PF requests a character from the user. The second PCALL argument is interpreted by the CLI to be the address of a prompt string for the user. No prompt is indicated by a 0 second argument. The PF will be PCALLed back with first result ´nextchar´ and second result the requested character.

´itsme´. The PF has determined that it may be on the correct path through the grammar and is asking the CLI to decide if there are other potentially correct paths. If the CLI decides the PF is on the only correct path, it will PCALL back with first result ´itsme´; otherwise it will PCALL back with first result ´nextchar´ and second result a character from the user.

´notme´. The PF has determined that it is not on the correct path through the grammar. Typically, the PF will not be PCALLED back. However, if there are no other paths through the grammar, or the user backs up, it will be PCALLED again.

´dosuc´. The PF has determined that it is on the correct path through the grammar, has completed any processing it needs to do, and is asking the CLI to do its successor in the grammar. After PCALLing with first argument ´dosuc´, this invocation of the PF will disappear. The second PCALL argument is interpreted by the CLI as follows. If it is nonzero, it is saved by the CLI and will be put in OPENPORT argument ´saveword´ when the PF is OPENPORTed for reasons ´terminate´, ´abortcmd´, or ´backup´. If the second PCALL argument is 0, the PF will not OPENPORTed for reasons

´terminate´, ´abortcmd´, or ´backup´; it will only be OPENPORTed for reason ´parsing´.

´backup´. The PF wants to back up the command. This could occur when the user types the backspace character numerous times.

´abortcmd´. The PF wants to abort the command.

A simple example is useful. The following PF sets a global to be TRUE. After completing its processing it PCALLs back with ´dosuc´ and 0, meaning that it is done and that it does not need to be invoked for reasons ´terminate´, ´abortcmd´, or ´backup´.

```
(simplepf) COROUTINE ( reason, instruction, accumulator
REF, argcount, arguments REF, saveword );

  PORT ENTRY EXIT PCALL;

  needconfirm _ TRUE;

  PCALL( dosuc, 0);

  END.
```

A more complex example is provided by the PF that generates a number to show to the user. Because it allocates storage, it must free it when the command is terminated, aborted or backed up.

```
(numberpf) COROUTINE ( reason, instruction, accumulator
REF, argcount, arguments REF, saveword REF );

  LOCAL number, cmlvalue REF;

  PORT ENTRY

    CASE reason OF

      = terminate, = abortcmd, = backup:

        freeblk( &saveword, $freespace); % free the block
        that the cml value was put into %

      ENDCASE;

  EXIT PCALL;
```

```
    % generate a number to show to user %

      number _ ackerman( 10 );

    % get a block from freespace to put CML value into %

      &cmlvalue _ getblk( 3, $freespace);

    % set up type of CML value %

      cmlvalue.vtype _ integer;

    % and length %

      cmlvalue.vlength _ 2;

    % put the number into the CML value %

      cmlvalue[1] _ number;

    % put a pointer to the value into the CLI accumulator so
    that it can be accessed by the grammar %

      accumulator _ &cmlvalue;

    % save the address of the block so that it can be freed
    upon terminate, abortcmd, backup %

      PCALL( dosuc, &cmlvalue);

    END.
```

During the specification of the command in which it is referenced, ´numberpf´ will first be OPENPORTed with reason ´parsing´. The CASE statement in the PORT ENTRY block will do nothing because reason is ´parsing´. When ´numberpf´ is PCALLed back, it will generate the number, create a valid CML value, put a pointer to it into accumulator, and PCALL that it is done and that the pointer to the CML value be saved. This invocation of ´numberpf´ will then disappear. When the command is finished, ´numberpf´ will be OPENPORTed again with reason ´terminate´ and ´saveword´ will have as its value the pointer to the CML value. If the command is aborted or backed up ´numberpf´ will be OPENPORTed with reason ´abortcmd´ or ´backup´. The CASE statement in the PORT ENTRY block will free the block. This invocation of ´numberpf´ will then vanish.

A final example illustrates a PF processing user input. The PF ´spchpf´, below, is invoked from the grammar with a single argument, a string. If the user types one of the characters in the string, ´spchpf´ will tell the CLI to continue processing down this path of the grammar tree. Otherwise, ´spchpf´ will tell the CLI that this is the wrong path. If the path through ´spchpf´ is the only path at this point in the grammar and the user mistypes, ´spchpf´ will read another character and again check if it is in its argument string. Spchpf will give the user a prompt of "SPCH" and will respond to ? with "Type a Special Character".

```
(spchpf) % Let the user type one of any of the characters
in the CML argument string%
COROUTINE ( reason, instruction, accumulator REF, argcount,
arguments REF, saveword REF );

  LOCAL

    cmlarg REF, % the argument passed in the grammar %

    spchstr REF, % the string of special characters %

    i, % loop index %

    char, % character the user typed %

    prompt REF; % the prompt string %

  PORT ENTRY EXIT PCALL; % do not care about reasons
  terminate, abortcmd or backup %

  % get the string of special chars %

    &cmlarg _ arguments; % cmlarg points to CML value
    passed from grammar %

    &spchstr _ cmlarg[2]; % spchstr points to string from
    CML value %

  &prompt _ $"SPCH"; % set up prompt string %

  reason _ nextchar; % want to get char from user %

  LOOP % getting characters from CLI %

    CASE reason _ PCALL(reason, &prompt := 0 : char) OF
```

```
        = nextchar : % process the next character checking to
        see if it is in spchstr %

          CASE char OF

            = ´?: % user wants a more verbose prompt %

              &prompt _ $"Type a Special Character";

            ENDCASE % check to see if char is in spchstr %

              BEGIN

              FOR i _ 1 UP UNTIL > spchstr.L DO

                IF char = *spchstr*[i] THEN % char in spchstr
                - tell CLI this right path through grammar %

                    PCALL( dosuc, 0);

              reason _ notme; % wrong path through the
              grammar %

              END;

        ENDCASE;

    END.
```

Let us assume that ´spchpf´ is to be used to determine whether or not the user has typed a digit. An appropriate reference to it in the CML grammar is

```
  ... spchpf( #"0123456789" ) ...
```

Upon reaching this point in the grammar, ´spchpf´ would be OPENPORTed and it would PCALL back to the CLI with reason ´extchar´ and supplying user prompt "SPCH". ´Spchpf´ sets prompt to 0 so that the user will only be prompted once in the event that it reads more than one character. The user would see "SPCH:" and might type a "2". The CLI would then PCALL to ´spchpf´ with reason ´nextchar´ and character "2". ´Spchpf´ would find the 2 in ´spchstr´ and PCALL back to the CLI with reason ´dosuc´ and second argument 0 meaning that it should not be OPENPORTED for reasons ´terminate,´ ´abortcmd´ or ´backup´.

If the user typed a "?" as her first character, spchpf would return the more verbose prompt "Type a Special Character" and return it with reason nextchar. If the user then typed a digit, spchpf would proceed as before.

If the user typed "x" as her first character after the "SPCH:" prompt, the CLI would return it to 'spchpf' as before. 'Spchpf' would not find it in 'spchstr' and would PCALL back to the CLI with reason 'notme' indicating that it is not on the correct path. In the CML example we are using, the path through 'spchpf' is the only path through the grammar at this point. As a result, the CLI will inform the user that an inappropriate character was typed, collect another character, and PCALL back to 'spchpf' with reason 'nextchar' and the typed character. This process will continue as long as the user types bad characters, ending only when the user types a digit or aborts the command.

A more complex example is the use of 'spchpf' in parallel with other CML elements. Consider the following CML:

```
spchpf( #"0123456789" ) <"digit"> / "WORD" <"letter">
```

Upon reaching this point in the grammar, the user would see a prompt of "SPCH/C:" because prompts are provided by both spchpf and the command word. If the user specifies the command word by typing "W", 'spchpf' will PCALL back with reason 'notme', the command word will succeed, and the user will next see the noiseword "letter". If instead the user types a digit, 'spchpf' will PCALL back with 'dosuc' and the user will next see the noiseword "digit".

Writing Parse Functions

Parse function writers should adhere to the following template, which delineates the general parse function form. Code is included in the template to meet contingencies encountered in the most complex parse functions. Simpler parse functions such as the PF 'simplepf', shown above, may omit much of this code.

```
(pfname)   % CL: ; one-line-description %
COROUTINE ( reason, instruction, accumulator REF, argcount,
arguments REF, saveword % => result %);

  % Parsefunction description

    FUNCTION

      none
  ARGUMENTS

    none

  RESULT

    none

  NON-STANDARD CONTROL

    none

  GLOBALS

    none

  %

% Declarations %

  LOCAL char, prompt REF;

PORT ENTRY

  CASE reason OF

    = parsing: % being invoked for first time during command
    %

    = terminate: % command is done, cleanup %

    = abortcmd: % command was aborted, cleanup and restore
    state %

    = backup: % command was backed up %

    ENDCASE;

EXIT PCALL;

reason _ nextchar;

&prompt _ % prompt string for user or 0 %;

LOOP

  CASE reason _ PCALL( reason, &prompt := 0 : char ) OF

    = nextchar: % examine at character from user %
```

```
        BEGIN
        CASE char OF
          = ´?: % user wants more verbose prompt %
            BEGIN
            &prompt _ % the verbose prompt %;
            REPEAT LOOP;
            END;
          ENDCASE;
        % decide whether PF is on the correct path through the
        grammar %
        IF % on the right path % THEN reason _ itsme
        ELSE % not on right path % reason _ notme;
        END;
      = itsme: % CLI agrees that this is correct path through
      the grammar %
        BEGIN
        % do processing %
        saveword _ % word of context to be saved or 0 %;
        PCALL( dosuc, saveword);
        END;
      ENDCASE;
  END.
```

Selection Parse Functions

Selection parse functions (SPFs) are used to generate selection entities not supported by the CLI because they are not of general use. These may be generated for example by collecting user characters and interpreting them in a certain way. As a result of this special function, SPFs interact with the CLI slightly differently than do other PFs.

SPFs are associated with a command word that is declared to be a SELECTOR in the grammar. For example, if a grammar contains the declaration

```
  DECLARE COMMAND "ENTITY" SELECTOR TYPEIN = typentity;
```

then 'typentity' is an SPF and need not be otherwise declared as a PF. An SPF is OPENPORTed when the command word with which it is associated is given as an argument to DSEL, SSEL, or LSEL. Thus, when the CLI executes the CML

```
  var _ "ENTITY" LSEL( var )
```

'typentity' is OPENPORTed.

The OPENPORT arguments to an SPF are:

'type'. The type of the selection. It may assume the following values, which are declared as external constants in the FE.

'typein'--The SPF is being OPENPORTed because it was declared as the TYPEIN part of the SELECTOR.

'typeaddr'--The SPF is being OPENPORTed because it was declared as the ADDRESS part of the SELECTOR.

'pointsel'--the SPF is being OPENPORTed because it was declared as the POINT part of the SELECTOR.

instruction'. Byte pointer to CML instruction in grammar.

accumulator REF'. Address of the CLI accumulator. The accumulator will contain a CML value which is the command word at OPENPORT time. SPFs should save the contents of the CLI accumulator at OPENPORT time as it may change before the CLI PCALLs back.

If the declaration for "ENTITY" is

```
  DECLARE COMMAND "ENTITY" SELECTOR TYPEIN = typentity ADDRESS
  = typentity;
```

then upon executing the above CML, 'typentity' would immediately be OPENPORTed two times--once with type ´typein´ and once with type ´typeaddr´.

Clearly the CLI may OPENPORT up to three SPFs for a selection, one for each of POINT, TYPEIN, and ADDRESS. The CLI then determines which of the three to further interact with by getting a character from the user and examining it. For example, let us declare "THING" to be

```
  DECLARE COMMAND "THING" SELECTOR POINT = pointpf TYPEIN =
  typepf ADDRESS = addrpf;
```

and use it as

```
  LSEL( #"THING" )
```

The user would see as a prompt "B/T/[A]:" meaning that she could bug a THING, type in a THING, or give the address of a THING by first typing the OPTION character to distinguish it from typing in a THING. Note that the CLI, not the SPF, supplies the prompt, which is based on whether LSEL, DSEL, or SSEL is being used and the SELECTOR declaration. If THING had been declared--

```
  DECLARE COMMAND "THING" SELECTOR TYPEIN = typepf ADDRESS =
  addrpf;
```

--the prompt would have been "T/[A]:".

Assuming again the first declaration of THING, the CLI determines with which SPF to interact. If the first character the user types is a COMMAND ACCEPT at a display terminal, the CLI will interact with ´pointpf´. If the first character is OPTION, the CLI will interact with ´addrpf´. Otherwise, the CLI will interact with ´typepf´.

Because a SPF does not supply a prompt, it is PCALLed back immediately with reason ´nextchar´ and the user character. It then interacts with the CLI by PCALLing for various reasons, such as a PF.

SPFs must put a valid CML value in accumulator and PCALL with the first argument ´dosuc´ and no second argument. Unlike PFs, an SPF should not free storage that it allocated for the CML value that it returns in accumulator. This storage will be freed by the CLI at the proper time.

An example of a typical SPF is shown below. It calls the FE routine ´gettext´ to actually make the CML value for it and put it in accumulator.

```
(typicalspf) % a typical SPF %
COROUTINE ( type, inst, accumulator REF);

  LOCAL

    saveaccum = accumulator, % save the value of the
    accumulator to restore later %

    char, % the user characters %

    reason;

  PORT ENTRY EXIT reason _ PCALL(: char);

  accumulator _ saveaccum; % restore the value of the
  accumulator %

  gettext(type, inst, &accumulator, FALSE, PORT, temp,
  reason, FALSE );

  PCALL( dosuc);

  END.
```

Built-in Parse Functions

The CLI contains several built-in PFs that may be accessed from a grammar in the same way as other PFs. These built-in PFs must be declared.

The following built-in PFs are described with their CML arguments and results.

NSW Parse Functions

fesetuser (userid, nodeprofile, userprofil, project, node =>)

This PF sets up the builtin CML variables, user profile and node profile for a particular NSW user.

feclruser (=>)

This PF removes all information about the current NSW user from the FE.

fegenusename (toolname => usename)

This PF generates a unique usename for a tool given the tool name.

fenewtool (toolname, usename, processhandle, grammarname =>)

This PF sets up the FE to run a new NSW tool. It gets the grammar for the tool and sets up a connection to the tool if necessary.

fersmtool (usename =>)

This PF allows a tool to be resumed whose use had been suspended by the user.

fedotermrule (usename =>)

This causes the termination rule of a tool grammar to be executed by the CLI. It is normally invoked just before feendtool.

feendtool (usename =>)

This PF removes a tool instance use from the FE. It also causes any connection between the FE and the tool to be broken.

Subsystem Parse Functions

fenewsubsys (subsystemname, grammarname =>)

This PF sets up the FE to run a new subsystem.

feendsubsys (=>)

This PF is invoked inside a subsystem to end it.

Miscellaneous Parse Functions

feforceupper (cmlstring => cmlstring2)

This PF returns as its result a copy of its argument in all upper case letters.

feterminate (=>)

This PF terminates the FE. It may only be called from the executive grammar (the first grammar the FE uses when it is started).

festwtp (windowtype =>)

This PF sets the type of the current window. Its single argument must be an integer that indicates a valid window type. For more information, see "The Virtual Terminal Controller", in FRONTEND SYSTEM DOCUMENTATION.

festtp (terminaltype =>)

This PF sets the type of the user´s terminal. Its single argument must be an integer that indicates a valid festtp type. See "The Virtual Terminal Controller", in FRONTEND SYSTEM DOCUMENTATION.

fecup (type, value, value2 =>)

This PF is used to change user profile parameters in the FE. The first argument indicates which parameter is to be changed, the second is its new value, and the third is a secondary value depending on the first two.

APPENDIX 6: CML Syntax

Syntax Notes

This appendix offers a formal description of CML syntax, described through the Tree Meta principles of alternation (indicated by the symbol /) and succession (denoted by juxtaposing elements). The syntax closely resembles CML itself, with the addition of the following symbols.

.ID	An identifier.
.SR	A quoted string.
/	Denotes alternatives. A/B means A or B.
%	Brackets comments.
()	Used for grouping to control precedence.
[]	Denotes optional elements.
'	Precedes literal characters.
"	Encloses literal strings.
#X	At least one occurrence of whatever X is.
#<Y>X	At least one occurrence of whatever X is, separated by whatever Y is.
$X	Zero or more occurrences of whatever X is.
$<Y>X	Zero or more occurrences of whatever X is, separated by whatever Y is.
.NUMBER	A string of digits representing a non-negative (decimal or octal) integer.

Formal CML Syntax

```
% PARSING RULES %
  % file definition %
    file = "FILE" .ID
      $condcomp $dcls $rule   % can have global rules or
      declarations %
      #(subsys) % must have at least one subsystem def %
      "FINIS";
      %---------

  % DCL definitions %

    condcomp = %conditional compilation switches%
      "SET"
      #<´,> ( .UID ´=
        ("TRUE" / "FALSE") )
      ´; ;

    dcls =
      "DECLARE"
        ( "COMMAND" "WORD"
          #<´,> ( .SR [ ´= .NUM ] [slctr] ) ´;
        / [dclattr]  #<´,> .ID   ´;
        / dclfunattr  #<´,> dclitem   ´;
        / "CONSTANT" #<´,> (.ID ´= .NUM ) ´;
        ) ;

    slctr =
      "SELECTOR"
        (´= biselectors
        / $(
          "POINT" ´= (.ID / (biselectors))
          / "TYPEIN" ´= (.ID / (biselectors))
          / "ADDRESS" ´= (.ID / "TEXT" / "CHARACTER")
          )
        );

    biselectors =  (
      "CHARACTER" /
      "WORD" /
      "VISIBLE" /
      "INVISIBLE" /
      "TEXT"  /
      "INTEGER" /
      "NUMBER" /
      "STRING" /
      "OLDFILENAME" /
      "NEWFILENAME" /
      "FILENAME" /
      "PASSWORD" /
      "CHARPOS" ["ITION"] );

    dclattr = % declaration attributes %
      ("VARIABLE"
      / "PARSEFUNCTION"
    / "GLOBAL" [ "VARIABLE" ] );

  dclfunattr = % declaration function attribute %
    "FUNCTION"
      ["PROCESS" ´= .SR [ ´, "PACKAGE" ´= .SR ] ´: ];

  dclitem = .ID
    ["OUT" "OF" "LINE" /
    "PSEUDONYM" ´= .ID ["OUT" "OF" "LINE" ]
    ];

% SUBSYSTEM definition %
  subsys = "SUBSYSTEM" .ID  "KEYWORD" .SR
    #(command / rule)
     "END.";

% command definitions %

  command =
    ( ( "COMMAND" .ID / .ID "COMMAND" ) ´= exp    ´;
    / "INITIALIZATION" rule
    / "TERMINATION" rule
    / "HELP" rule
    / "REENTRY" rule
    ) ;

  rule = .ID ´= exp ´; ;

% expression definition %

  exp = .#<´/> subexp;
  subexp = #factor ;
  factor =
    ´( exp ´)
    / ´[ exp ´]
    / term;

% terminal nodes for compiler %

  term =
    (subname/ confirm/ answer/ optchar/ feedback/
    recognition/ variablecw/ loop/ conditional/ showuser/
    abortcmd/ helpresume / tparam);

  conditional =
    "IF" ["NOT"] param
      [ %binary relation%
        ( ´=
        / ">="
```

```
          / "<="
          / "< "
          / "> " )
          ( idornum / "NULL" )
        ] ;

  idornum =
    .ID
    / .NUM
    / bivar1
    / bivar2;

  showuser =
    "SHOW" ["CONFIRM" ] ´(  param  ´);

  subname =
    (.ID / bivar1)
      [
      ( ´( $<´,>pfparam ´) )  % parse function invocation %
      / ( [inlinopt] ´( $<´,>param [funresults] ´) )  %
      function invocation %
      / (´_ param )  % assignment statement %
      / (´: ´_ param )  %append statement %
      ] ;

  inlinopt =
    ´[ ["IN" "LINE" / "OUT" "OF" "LINE" ] [ .ID ] ´];

  funresults =
    "->" #<´,> (.ID / bivar1);

  recognition = keyword / builtinrec;

    keyword = .SR [ ´! #qualifier ´! ];

    builtinrec =
      ( ( "SSEL"
        / "DSEL"
        / "LSEL" )
      ´( param ´) );

  variablecw =
    "CW" ´:
      ( .ID
      / bivar1
      / bivar2
      ) ;
```

```
feedback =
  ´< ["..."] .SR ´>
  / "CLEAR";

confirm = "CONFIRM" ; % call routine to terminate cmd %

answer = "ANSWER" ;  % call routine to process yes/no
answer %

abortcmd = "ABORT" [ ´( param ´) ];

helpresume = "RESUME" [ ´( param ´) ];

optchar = "OPTION" ;

loop = "PERFORM" .ID "UNTIL" ´( exp ´);

qualifier = ("L2" / .NUM );

param = tparam / factor;

pfparam = param / addrparam;

addrparam = %pass grammar variable address to parse
function%

  ´$ (.ID / bivar1 / bivar2) ;

tparam = "TERMINATORS";
  ´# .SR
  / .NUM
  / "NULL"
  / "TRUE"
  / "FALSE"
  / (bivar1 / bivar2);

bivar1 = "TERMINATORS"; % assignable builtin vars %

bivar2 =  % non-assignable builtin vars %
  ("TERMCHAR"
  / "USERID"
  / "PROJECT"
  / "NODE"
  / "DISPLAY"
  / "TYPEWRITER"
  / "LINEATATIME"
  / "WINDOW"
  / "HELPCODE"
  / "HALFDUPLEX"
/ "CURRENTTOOL"
/ "BREAKPOINT"
/ "RESULT1"
/ "RESULT2"
/ "RESULT3"
/ "RESULT4"
/ "RESULT5"
/ "RESULT6"
/ "RESULT7"
/ "RESULT8"
/ "ACTIVETOOLS"
/ "TOOLS"
) ;
```

APPENDIX 7: HELPCODE and HELP Rules

This appendix describes the use of the built-in variable HELPCODE and the other HELP rule facilities of CML. It complements the section on the FE externally callable ´help´ procedure and the three possible kinds of HELP rules, described in Appendix 4 "Externally Callable Procedures in the Frontend". The reader should be familiar with the CML syntax for specifying HELP rules described in the GUIDE TO THE CML AND CLI.

A CML HELP rule is invoked when a Backend execution function calls the FE externally callable ´help´ procedure. The particular HELP rule invoked depends on the value of ´helptype´, the first argument in that procedure. The mechanism is discussed in detail in Appendix 4.

Writing CML HELP Rules

CML HELP rules are similar in structure to other CML rules. A HELP rule will typically do three things:

Give the user some information. This may supplement information supplied through the ´help message´ argument of the ´help´ procedure (see Appendix 4). For example, noise words in the HELP rule may specify a question to the user.

Solicit some information from the user. This may be done by having the user specify some command words or input a selection. The rule may use the CML built-in HELPCODE to decide what type of information to get from the user. The value of HELPCODE is set by the value of the argument ´helptype´.

Return the collected information. This is with a RESUME(var), where var is a CML variable containing the information. The contents of this variable is returned to the externally callable ´help´ procedure, which returns it to the Backend execution function.

The HELP rule for a grammar may be specified by the syntax:

```
HELP rulename1 = ...;
```

where "..." indicates the normal syntax for a CML rule. The rule will be executed whenever the FE ´help´ procedure is called with a first argument value of 3.

A specific rule may be defined for particular execution function calls in the grammar by the following syntax:

```
...xroutine [rulename2] (arglist...)...
```

Here rulename2 specifies a CML rule defined elsewhere in the grammar. It will be executed if a call is made from the Backend on the FE ´help´ procedure with a first argument value greater than 100 during the processing of the particular instance of xroutine.

Examples

Examples of the use of the FE Help facilities are presented below. The first example is from the grammar for the NSW Works Manager. It is interesting because its Backend is not written in L10. The rule for the delete command calls the execution function ´wmdelete´, with an explicit HELP rule name (´switch´) indicated in square brackets. If, in the Backend, a call is made on the FE procedure ´help´ with a first argument greater than 100, the rule ´switch´ will be executed to collect new parameters as specified by the value of the first argument. The value of that argument will be assigned to HELPCODE, which is examined in the ´switch´ rule. At the end of the rule, a ´RESUME´ is executed.

```
CONSTANT %result codes%
  gettext = 101 ,
  getnewfn = 106 ,
  getoldfn = 107 ,
  getinteger = 108 ,
  getpasswd = 110 ,
  getlist = 111 ,
  note = 113 ,
  yesorno = 114 ,
  getofnorn = 115 ,
  getproj = 126 ,
  getnode = 127 ;
.
.
.
delete COMMAND =
  "DELETE" <"File named:"> ofn _ LSEL(#"FILESPEC")  CONFIRM
  wmdelete [switch] (USERID, ofn, TRUE)  CLEAR
  ;  % END delete %
.
.
```

```
switch =
  ( IF  HELPCODE = gettext  resp _ LSEL(#"TEXT")
  / IF  HELPCODE = getnewfn  resp _ LSEL(#"ENTRYNAME")
  / IF  HELPCODE = getoldfn  resp _ LSEL(#"FILESPEC")
  / IF  HELPCODE = getinteger  resp _ LSEL(#"INTEGER")
  / IF  HELPCODE = getlist  resp _ NULL  intlist
  / IF  HELPCODE = note  resp _ NULL
  / IF  HELPCODE = yesorno  resp _ ANSWER
  / IF  HELPCODE = getofnorn
    ( resp _ LSEL(#"INTEGER") / OPTION resp _
    LSEL(#"FILESPEC") )
  / IF  HELPCODE = getproj
    resp _ LSEL(#"NSWPROJECTNAME")  proj _ resp
  / IF  HELPCODE = getnode
    resp _ LSEL(#"NODENAME")  node _ resp
  / IF  HELPCODE = getpasswd  resp _ LSEL(#"PASSWORD")
  )  CLEAR
  RESUME ( resp )
  ;  % END switch %
```

The second example is taken from the ARC debugger, DAD. This illustrates a typical L10 Backend call that results in a call on the Frontend 'help' procedure. As is also the case in the NLS Backend, procedure calls on the Frontend are made through a call on a procedure in a "middle-end", which is responsible for packaging arguments to the target procedure, dispatching to the procedure, and receiving and decoding any returned values.

The grammar for the DAD debugging tool has a HELP rule that will be called if the first argument to the 'help' procedure is of type 3. Note that it is essentially a null rule and simply returns TRUE. More interesting code is executed if the type specified by the first argument is greater than 100. As in the first example, the call on the execution functions must indicate the rule to be executed. If the DAD Backend calls the FE 'help' procedure with a first argument of 2, the help grammar associated with the executive would be entered. (DAD does not use this feature.)

```
% The default DAD HELP rule: %
  HELP dghelp =
    IF HELPCODE = ghlpcode
      newevent _ TRUE
      RESUME ( TRUE )
    ;
```

% Note that the move command rule calls Backend execution functions and specifies that the rule "nvlrul" will be called if the argument to the FE "help" procedure is greater than 100. %

```
  movrul COMMAND = IF nvltrm   nvtemp _ nvltrm   nvltrm _
  FALSE
    ( IF #" " = nvtemp %TAB%   xdtab[nvlrul]()
    / IF #"#" = nvtemp    xdpound[nvlrul]()
    / IF #"
           " = nvtemp    xdlnfd[nvlrul]()
    / IF #"^" = nvtemp    xdupar[nvlrul]()
    );
```

% In the Backend, a call on the frontend "help" procedure is made in the procedure "pasnstr" %

```
  .
  .
  .

  % Set up arguments for the FE help procedure.  The value of
  cmlnwvalue, a DAD global, will specify the type of help.
  In DAD it is always greater than 100, so the appropriate
  HELP rule specified in the execution function call would be
  executed. %

    *locstr* _ *astr*, "   _ ";

    #alist# _ cmlnwvalue, *tstr*, *locstr*;

    IF extcall( smbox, $"HELP", 2, $alist, $rlist ) THEN
      BEGIN
      intype _ ELEM #rlist#[1];
      instype _ ELEM #rlist#[2];
      IF ELEM #rlist#[3] THEN *nstr* _ *[ELEM #rlist#[3]]*
      ELSE nvalp _ FALSE;
      END
  .
  .
  .
```

% nvlrul was specified as the HELP rule for all the execution function calls in the move command rule exhibited above. It will be executed if the first argument to the frontend help procedure is greater than 100. The value of that argument will be assigned to the built-in HELPCODE. %

```
nvlrul =
  IF HELPCODE = newvalue
    intype _ FALSE   instype _ FALSE   nvltrm _ FALSE
    nvllst _ NULL   nval _ FALSE
    ( CONFIRM
    /
      [ OPTION <"input mode"> inptyp <"new value"> ]
      nval _ LSEL(#"NVALUE")
    )
    nvllst :_ intype   nvllst :_ instype   nvllst :_ nval
    RESUME( nvllst );
```

APPENDIX 8: Standard Control Characters for Standard Functions

The following control characters perform special functions for the user and are interpreted by the CLI.

<CTRL-A>:	Backspace character or parse state (also represented by <CTRL-H>).
<CTRL-B>:	Repeat character. When used to confirm a command or terminate a literal typein, it causes the current command to be repeated with new arguments. Represented in CML by REPEAT.
<CTRL-C>:	Not used (used by TENEX).
<CTRL-D>:	Command Accept. Used to confirm commands or terminate literals. Represented in CML by CONFIRM.
<CTRL-E>:	Not used.
<CTRL-F>:	Not used.
<CTRL-G>:	Not used (might become the <CTRL-Z> function defined below).
<CTRL-H>:	Backspace character or parse state (also represented by <CTRL-H>).
<CTRL-I>:	Tab.
<CTRL-J>:	Linefeed.
<CTRL-K>:	Not used (used by MSG debugger).
<CTRL-L>:	Invokes the NSW debugger.
<CTRL-M>:	Carriage return.
<CTRL-N>:	Not used.
<CTRL-O>:	Aborts an unwanted typeout or command execution in the Backend.
<CTRL-P>:	Backspace typein. Used to delete the entire typein of an argument to a command.
<CTRL-Q>:	Invokes semantic help.
<CTRL-R>:	Retypes literal typein.

<CTRL-S>:	Invokes syntax printout of part of a command or all commands in a tool or the EXEC.
<CTRL-T>:	Not used (used by TENEX).
<CTRL-U>:	Option. Used to access infrequently used or dangerous parts of commands. Represented in CML by OPTION.
<CTRL-V>:	Literal escape. Allows entry of control characters in literal typein. The character following the <CTRL-V> is accepted as normal input.
<CTRL-W>:	Backspace word or parse state.
<CTRL-X>:	Abort current command specification.
<CTRL-Y>:	Not used.
<CTRL-Z>:	Escape to NSW EXEC. Also aborts current command specification.
?:	Causes the CLI to type current alternatives, except when ? is encountered in the middle of literal typein.
SPACE:	In some Terse command recognition mode, this is used to specify second level command words. Can also be used to terminate command words in DEMAND recognition mode.
ESC:	May be used in DEMAND recognition mode to complete commandwords.

APPENDIX 9: User Profile Users' Guide

Introduction

You can alter how you interact with the system you are using to fit your own equipment, use patterns, and style by specifying the parameters controlled by the User Profile tool.

The User Profile tool manipulates a data structure (called the User Profile) that controls the behavior of the system as the user sees it. Through commands in the tool you may adjust the behavior of the system to suit your individual preferences. Note that the User Profile influences only tool independent attributes and applies therefore to all tools and subsystems.

The User Profile can be divided into two major parts: the first allows you to tell what tools and subsystems you want made available to you; the second allows you to control how you want those tools and subsystems to appear. The first category includes names of programs and tools that you may have made available to you and some instructions on how you want to start your session with the computer. In the "system appearance" category fall control characters, feedback settings, heralds, prompts, and recognition modes.

The complete syntax of each command and a description of what it does is provided below. The following syntax conventions are used:

Command words are capitalized.

Noise words are in parentheses.

Alternatives and/or parameters to be collected from the user are in all caps, with an explanation following the command syntax.

CONTENT indicates that the system expects the user to type in something at that point.

OK means command confirmation (COMMAND ACCEPT or <CTRL-D>).

Commands in the User Profile tool

```
Control (characters for terminal) DEVICENAME OK
(function) FUNCTION (character(s)) CONTENT (echo as) CONTENT OK
```

DEVICENAME=
- Execuport
- Imlac
- Lineprocessor
- Nvt
- Tasker
- Ti
- Tty

FUNCTION=
- BC
- BW
- BS
- CA
- CD
- IGNORE
- LITESC
- OPTION
- RPT
- SC
- SW
- TAB

The User Profile "Control" command enables you to assign new characters (visible or invisible) for control characters and a new echo to serve a specific function on a specific terminal device. Note that this command always replaces the old entry with the new one. These control characters may differ for different devices. The user may specify more than one character for a specific function (in which case each of the characters will serve the same function) and may also assign an echo string to be typed out when a control character key is hit.

Default Control Characters

The table below describes all the control functions and the characters assigned to them by default. The default echo is always NULL.

FUNCTION	CHARACTER	DESCRIPTION
BC	^H	Backspace Character
BW	^W	Backspace Word
BS	^P	
CA	^D	OK, COMMAND ACCEPT

CD	^X	COMMAND DELETE
IGNORE	(-)	Ignore from input stream
LITESC	^V	Literal escape character
OPTION	^U	OPTION character
RPT	^B	REPEAT
SC	(-)	Shift character
SW	(-)	Shift word
TAB	^I	TAB character

Feedback Mode FBKMOD OK

FBKMOD = Verbose / Terse

The User Profile "Feedback Mode" command controls the expansion of command and noise words. Two modes are available: Verbose and Terse. Terse mode suppresses all noise words; Verbose, the system default, types out everything.

Feedback Length CONTENT OK

The User Profile "Feedback Length" command allows you to limit the length of command and noise word echoes when your Feedback mode is Terse. For CONTENT type in a number from 1-255.

Feedback Indenting (to be) CONTENT OK

The User Profile "Feedback Indenting" command allows you to control the indentation preceding the command line. For CONTENT type in a number from 1-15.

Herald Mode HRLDMOD OK

HRLDMOD = Verbose or Terse

The User Profile "Herald Mode" command allows you to control the way heralds are displayed. Two modes are available: Verbose and Terse. In the Terse mode no herald is typed (the user assumes he knows where he is); in the Verbose mode heralds are typed.

Herald Length CONTENT OK

The User Profile "Herald Length" command allows you to control the number of herald characters typed by the system. For CONTENT type in a number from 1-30.

Prompt PROMPTMODE OK

PROMPTMODE = Full, Off, or Partial

The User Profile "Prompt" command allows you to control the way you will be prompted. You may set prompts to one of three modes: Full (the system default) shows all prompting; Partial eliminates prompting for options; Off turns off all prompting.

Recognition (mode) RMODE OK

```
RMODE = Anticipatory
   or   Demand
   or   Fixed
   or   Terse (secondary mode) RMODE2
RMODE2 = Anticipatory, Demand, Fixed, or Terse
```

The User Profile command "Recognition" allows you to change your recognition mode. Your choices for RMODE are Fixed, Anticipatory, Demand, and Terse. With Terse mode you can make a further choice for your secondary mode of recognition. The Reset Recognition command will set your mode to Terse, Terse.

Recognition Modes Description

You can adjust the number of characters you must type to have the system recognize the command you are specifying. Four recognition modes (RMODE, above) are available:

Fixed

All command words will be recognized after you type the first three letters. You may NOT type more letters, because they will go into the next field.

Anticipatory

Each command word will be completed after you type enough letters to uniquely define it. You may NOT type more than the minimum number of letters, since the command word will have been recognized and the extra characters will go into the next field. For example, "Se" is enough to recognize the command Set. If you type "Set," the "t" would be read as the next part of the command. In NLS, the command would be interpreted as "Set Tenex".

Demand

Each command word will be recognized after you've typed enough characters to define the command word uniquely. If you are set for Demand recognition, you force a word's recognition by typing either <ESC> (Escape character or Alt Mode) or a space <SP>. It won't take off into the next field until you do this.

Terse

One command word is recognized for each single character. To recognize other commands beginning with the same character, you must first type a space to make these commands available in a secondary mode. Four secondary recognition modes are available in Terse after the space. These secondary modes work like they do when they are primary modes.

Secondary Modes

Terse Secondary mode (Terse Terse) works like Terse Anticipatory except that you can only specify second level command words--i.e., those preceded by a space. With the rest of the secondary modes, after you type the space you have the choice of all words beginning with that letter.

Include (program/tool) CONTENT OK

The User Profile "Include" command allows you to name a tool, program, or subsystem you wish to have included with your system. The default does not supply any tools. For CONTENT type in any legal program or tool name. Note: The Userprofile tool will not make any attempt to verify the existence of a tool or program named by the user, nor will it check the access rights of the user to the named tools.

Entry (program/tool) CONTENT OK

The User Profile "Entry" command allows you to specify a program or tool to be invoked as soon as you log into the EXEC. The system default is no entry program. Note: The

User Profile tool will not make any attempt to verify the existence of a tool or program named by the user nor will it check the access rights of the user to the named tools.

Exclude (program/tool) CONTENT OK

The User Profile "Exclude" command allows you to exclude any program or tool previously included in the default list. Exclusion of the entry program or tool will leave the entry specification empty (system default).

```
Reset All OK
Reset Control (characters for terminal) DEVICENAME (function)
FUNCTION OK
Reset Control (characters for terminal) DEVICENAME (function)
FUNCTION All OK
Reset Control (characters for terminal) All OK
Reset RESETPARAM OK

  RESETPARAM=
    Default (tools and programs)
    Entry (tool)
    Feedback
      Mode
      Length
      Indenting
    Herald
      Mode
      Length
    Prompt (mode)
    Recognition (mode)
    Startup (commands branch address)
```

The User Profile "Reset" command will reset the specified property to the system default. (The choices for DEVICENAME and FUNCTION are listed above, under the "Control" command.) The following commands are special cases:

When using "Reset Control (control characters for terminal)" to reset the control characters definition, the user must specify the terminal device name or "All" for all devices. A single control function for a specific device cannot be reset. (The user may, however, replace the specific entry with a default entry using the "Control Character" command).

"Reset Entry" will define an empty entry tool but will leave the previous entry tool included in the default program/tool list.

"Reset All" resets everything but the control characters.

```
Show All OK
Show Control (characters for device) DEVICENAME OK
Show SHOWPARAM OK

  SHOWPARAM =
    Default (tools and programs)
    Entry (tool)
    Feedback (mode)
    Herald (mode)
    Prompt (mode)
    Recognition (mode)
    Startup (commands branch address) OK
```

The User Profile "Show" command displays the current content for the requested parameter. (The choices for DEVICENAME are listed above, under the "Control" command.)

APPENDIX 10: How to Write a Help Description File

Background--Help and Other Frontend User Aids

Typing the Help button or using the Help command available for all tools provides the user with a description from the current tool's Help description file and places him in a repeating Help command. Questionmark and <CTRL-S> are technically not part of the Help function but rather a function of the Frontend command parser. This section contains background on Frontend Help services.

Questionmark: (?). If the user types a questionmark at any point in a command, the Frontend will type the command alternatives available to him at that point. After the list has printed he is left at the point where he typed questionmark.

<CTRL-S>: Show Command Syntax. Typing <CTRL-S> will provide the command syntax for the command currently being specified.

<CTRL-Q>: HELP Button. Typing the Help button or <CTRL-Q> at any point in a command provides a description about what the user is doing and places him in the Help command, where he can also ask for the meaning of other terms.

Help TYPEIN/OK. The Help command provides the most complete online information about the tool being used. By typing in any term followed by a confirmation, the user will see a description of the term, or information pointing out different uses of the term which will lead him to the description he is seeking. The Help command will then be ready for another term, the number of a "menu" item followed by a confirmation, < , or ^ (see below). (Capitalization does not matter.) Command Delete <CD> or <CTRL-X> will terminate the Help command.

A menu is a numbered list of related subjects that may follow a description in the Help command. Typing a number followed by confirmation will show the explanation named.

Once in the Help command users can move around in the "structure" of the Help data base. Any time after the first description prints, the user can type < (left anglebracket) followed by y (for yes) to see the previous view indicated or n (for no) to choose a view before that. If the user types ^ instead of < , he will go "up" instead of "back". Going "up" lets him move up a level in the Help data base file structure. This means that the user will probably see a more general term and its description.

Finding a Description

The words typed by the user in the Help command or descriptions of the command states sought by use of the Help button (or <CTRL-Q>) are matched to description names (statement names) in the Help description file. The various description file hierarchies are searched in a prescribed order until all possibilities have been exhausted.

Named and Unamed Descriptions

The name of a description in a Help file is the first word (with no preceding characters) of the description. A "named" description may be addressed directly by the user by typing the word in the Help command. If any non-alphabetic character precedes the first word of a description (including invisible characters such as space) then it is "unnamed". When you want to "unname" a description place a space in front of the first word.

Search Algorithm

After a term (one or more words) has been specified by the user, it is matched to description names in a prescribed order. The "first" search, described below, applies both to the first word in a multiple word term and to a single word term. When the first word is found, the search continues for the second term, or, in the case of a single word term, the description for that word is displayed.

The First Search for a Term

If the user is already in the Help command when he types in a term, the branch where he is currently located is searched for an occurrence of the description name matching the first word typed. If it is not found and there is a link in the current statement, the branch defined by that link is searched. (This may be across files.) If it is still not found the current file is searched from the beginning to the end. If the user is just entering Help the file for the tool currently being used is searched first.

Index Files

Tool builders may want to have more than one data base, with connections (links) between them. For example, you may want to write a general data base containing information pertinent to a number of tools, with links from this file to the files for each tool. This general data base is called an index file. If, during a search, the first word is not found in a tool data base, the message "searching index" is typed to the user indicating that the index file is being searched.

The index file is specified in statement 1 (preceded by a percent sign) of the tool file in the exact form of "% index in <link to file>" (without quotes) (The "i" in "index" may be upper or lower case.)

Previous Files

If the term was not found in the index file, any files previously viewed by the user in the current Help session are then searched.

Searching for Succeeding Words in a Term

After the first name has been found, searches for any succeeding names specified in a multiple word term are limited to the branch designated by the preceding word. If the next name is not found, and there is a legal help link in the statement addressed by the first word then the branch addressed by that link is searched. If the next name is not found there, a view of the description defining the preceding name is displayed and the succeeding word is typed followed by a questionmark indicating that it was not found.

Searching for Duplicate Names

If the user re-specifies the same word twice in a row, it is assumed that he was not satisfied with the description at the current location and wants a higer-level description. Therefore the next higher search from the last search is attempted. For example, if the current description was found by searching a tool or subsystem description file, when the user re-specifies the same word in the very next round, the index file is searched. Descriptions at higher levels are more general and usually point down to descriptions with the same name at lower levels. The user can usually go directly to a specific level description by specifying the tool name in front of the term he wishes to see.

Stopping a Search

The user can type <CTRL-O> at any point to stop a search and receive the message "search stopped on" followed by the word currently being searched indicating the search was stopped before the name was found.

How to Write a Description File

This section describes the contents of a Help description file and lists conventions to follow in creating and maintaining a Help file. Some of the conventions are required by the software, others are useful for maintenance. The descriptions assume you understand NLS terminology.

Structure and Contents of a Description File

A description file for a tool or subsystem contains descriptions of the commands available in that tool or subsystem and the definitions of any special terms needed to use the commands. It may also describe procedures for using the commands, or provide scenarios for using commands to perform specific tasks.

The name of the description file will be the same as the name handed to the Frontend along with the tool grammar.

The typical high level organization of a tool or subsystem description file is as follows: the "top" level description which briefly defines the tool/subsystem and describes what the user can do with its commands; the link to its index file; a "First Searches" branch for duplicate names and spelling variations (may be in statement one with the index file link); an (optional) "how to" branch that defines special terms and tells how to use the commands; and last, a "commands" branch that lists and describes all of the commands in the tool. In most cases you will want to follow this organization, but it is not mandatory.

In a file, this organization looks like the following:

```
  0  Name of tool/subsystem
        (brief description)
  1  % First Searches Branch  (Index in <FILENAME,>)
  2  how to use the ... tool
  3  commands in the ... tool
```

"Top" Description at the Head of the File

The "origin" or first statement of the description should contain a brief description or "functional definition" of the tool or subsystem. The first word of this statement (with no preceding characters) must be the name of the tool. This statement and the first line of each un-commented statement one level below it are what the user sees when he hits the HELP button before typing any commands or if he confirms the Help command without entering any term. It may also appear if he hits the Help button while in a command state with no other valid Help description.

First Searches Branch

The "first searches" branch contains descriptions that 1) have no logical place in the structure of the description file, 2) act as an index to two or more occurrences of the same name or description, 3) are ambiguous, and 4) are alternate spellings or synonyms. Placed at the beginning of the description file, this branch is preceded by a % to prevent it from being a visible menu item under the tool description statement (see comment). It should have the statement name "First Searches". Keep the branch in aphabetical order to prevent duplication.

"How to" Descriptions

"How to" descriptions define special terminology, explain confusing or cryptic commands, and/or tell how to do specific tasks that can be accomplished with the tool in language easy for the user to understand. They may also provide specific scenarios for the user to follow. If your file is to contain how to information, name a branch "how to ..." and insert the information in substatements below it. The first line of each description should give a clear idea about what that statement explains and make sense to the user as a menu item. In most files the "how to" branch precedes the commands branch, both of them one level below the "top" description.

Commands Description Branches

The commands branch is the last branch one level below the "top" description (i.e. tail of the plex whose source is the top description). Insert a statement named "commands in the..." followed by a general description of the functions of the commands in that tool. In statements below this place command descriptions for every command that appears when you type ? in that tool. The commands should be in alphabetical order by the first command word. The first statement describing any command should have as its first word (with no preceding characters) the command word for the particular command for which it is a description (see below). Each command word always has only its initial letter capitalized to indicate that it is a command word.

Syntax: the First Line

The top line of each command description contains the syntax for that command or command part. This is terminated by a carriage return. The exception to this occurs when the syntax will not fit in the first few lines. In that case you may break up the command syntax in the most logical places with carriage returns, or make it a menu item named syntax. If you are writing about families of commands--e.g., commands with the same command verb, but different nominals--the statement name will not always be the same as the first word of the command syntax. The example below is typical. When the statement name is not the first word of the syntax description, the syntax follows two spaces from the colon following the statement name.

```
Show...
   Directory:  Show Directory (of) CONTENT/OK
   [(opt:)DIROPT] OK
   Disk:  Show Disk (space status) OK
   Return:  Show Return (ring) OK
   File:  Show File...
      Default:  Show File Default
      (directory for links) OK
      Modifications:  Show File Modifications (status) OK
      Return:  Show File Return (ring) OK
```

Function Description

Immediately following the carriage return which terminates the command syntax in the first line is the function description. The first line of the function description is indented two spaces. The first words of each command description contain the name of the tool (or subsystem) followed by the name of the command, thus:

```
Insert STRUCTURE (to follow) ...
  The Base command "Insert STRUCTURE" ...
```

The tool or subsystem name should be mentioned because the user may not know to what tool/subsystem a command description applies.

Multiple Command Words

Some commands may contain multiple command words to specify a tree of alternatives (i.e. a single command verb followed by several possible nominals). These alternatives stemmming from the same initial command word(s) are described in substatements below the initial command. This is illustrated in the example above under syntax.

Other Command Description Substructure

You may also wish to include statements beneath your command descriptions containing the following information:
example
syntax
effects
special terms
parse-functions
They should be placed at the end of the branch so as not to be confused with choices in the command.

Parse Function Descriptions

In general, there are a few built-in selections, rules or, parse functions which occur in many commands. These are given names always represented in all upper-case and described in only one place. For example the built-in selections LSEL, DSEL, and SSEL are called CONTENT, DESTINATION, and SOURCE and are defined along with other parse functions under a general description named VARIABLES in a "top-level" (index) description file that is common to all tools or subsystems using these functions. Thus each subsystem description file is augmented by the top-level file containg commonly used terms.

Substructure Rules

The basic rule for structuring description files is to follow some logical order of classification. Look at your file from several views; examine the relationship of statements to those around it; and check the flow of the file from top to bottom. Although the presentation of the information may be choppy, the ideas should follow in logical progression.

Menu Items

The substructure of any description (menued items) should contain only items classified by the name of that description. The items menued below a description should be in some logical order or in an order suggested by the description itself. Do not place (under statement n) menu items containing links to items which in turn have menu items which point to statement n. This is unsettling to users who think of the menu items as logical substructure classified under the branch node.

The first line of a description cannot be more than 64 characters long and should be terminated by a carriage return if there is a second line. It should make sense to the reader when seen as a menu item.

Because long menus are time-consuming for the user, it is generally a good idea to avoid using them for simple links to terms that already appear in the description they follow. For example, you would not menu a link to a single word term in the same file that clearly stands out in the description. You would, however, menu a link to a multiple word term located in another file that the user would not recognize as a possible typein. Using "See" rather than substructure also saves space and time.

Predecessors and Successors

Links to predecessors and successors to a description statement should not be included in the substructure of the description; instead, place them in a "See" list along with other terms of reference for that description and terms not logically classified by that node. In this way, only items properly classified by a branch node will be in its substructure and yet the surrounding items can be referenced.

Referencing: Links, "See" and "See also", and Compare

Links

If descriptions are written properly, you can avoid much redundancy by linking from one concept to another. Links may point to concepts in the same file or to concepts in other files, particularly index files. This is crucial to effectively update the Help descriptions when changes are made in the system. Description files containing links take on the qualities of a network.

Words in links are always all lower case to facilitate substitutions. Link delimiters everywhere must be ##< for the left and >## for the right. Only one link is taken. If you link to a statement that contains a link to another statement, the second statement you link to will not appear. The second link will not be taken--it will look to the user as if nothing is there.

There are several possible fields within a link though usually only a description name is used surrounded by delimiters. Internal fields are seperated by a space as follows:

##[QSPECS]<filename, name1 !name2 !name3>##

Some examples of links are: ##<editing>## ##<insert !string>## ##[C]<publication, directive !blank !gyes>##

QSPECS: (query viewspecs). QSPECS are special viewspecs enclosed in square brackets and located between the left double pound sign and the left link delimiter of a link. These are seldom used. They are mentioned here for completeness:

N = only menu "number" items, then ask "more?".

C = Columnate the names of the menued items: text associated with the menu items is not displayed until the menu item is chosen.

File name. File names are included only in links to a different file. If you are linking to a directory other than the default directory for links, the file name must be preceded by the directory name.

Name field. For single word terms only one name is required. For multiple word terms the first name defines the branch to be searched for the remaining name(s). Second name elements in a link must be preceded by an exclamation point. This will not cause the link to work like the second name searches in the Help command, but it will limit the search to the branch defined by the first name. You can include more than three name elements after the search field, but this is not likely to be necessary.

Illustration of Link Searching

For the link ##<filename, name1 !name2 !name3>## the following search is made:

```
  < FILENAME >
     name1
       name2
         name3  *this statement will be displayed*
```

Statements Containing Links

Statements that are to be menued but contain only a link (and brief text to tell the user what he will find) should have preceding text on the first line and the link on the second line thus:

```
Insert STRING command
##<insert !string>##
```

This causes an unfortunate wasted blank line but without following this convention the user would have to wait for the link to be searched and the first line of the addressed node displayed before he could see the rest of the menu.

"See" and "See also:"

Use "See" or "See also:" (no quotes) to any reference term that a) is not logically classified by the name of the description, b) disturbs the order of the menued items, or c) is not clearly mentioned in the description such that the user would recognize it for a term he could type in. "See" is used to point to primary concepts, "See also:" to secondary concepts.

Conventions for See and See also:

These references are usually located at the end of the node. They are treated as complete sentences with the understood subject "You": capitalize the first letter of the sentence, terminate with a period. A colon follows "also" but not "See". The terms being referenced should have their normal capitalization and be separated by commas. When you want to specify what information the reference points to and your sentence does not begin with "See", adapt conventional capitalization and punctuation.

Compare

In some cases you may wish to have the user compare the term you are describing with another term. To do so, use "Compare" in the same way that "See" is used to refer the reader to the term.

Backlinks

References (links and terms following "See" and "See also") should be manually "backlinked". This is done by going to the location referenced by the "See" or link and inserting a statement as the tail of the substructure, unless a backlink statement already exists. The backlink statement should follow this convention:

```
          % backlinks:  <...>, <...> etc.
```

where the text within the delimiters points to the statement containing the link or "See". Backlinks for references to locations in different files have first priority.

Pointing to an Unnamed Statement in a Backlink

If the place of reference you wish to point to is in an unnamed statement, use viewspecs or infileaddress elements in your backlink to point to it. For example, if the reference is an unnamed statement containing a link, you could show it by pointing to the statement above it and turning on viewspecs "ebt", or by using the address element ".d". If you are sure the original SIDs will never be renumbered, you can link to the SID. When the place of reference is difficult to find, you might want to include a comment following your backlink to help locate it.

Duplicate Names

Descriptions with duplicate names can exist in the same description file, except when the duplicate name is the same as the first word of a command description. (In this case you cannot duplicate, because the Help button will not find the command's description.) To reach a duplicate name, you must know which comes first and the name of a unique branch in which the duplicate statement is located. Duplicate names should be avoided whenever possible because the user must type in two or more words to reach the term in the Help command, and as a description writer you must remember if it needs two (or more) words when you want to point to it. If it is unique, one word is all that is required.

Choices for Handling Duplicate Names

If you cannot get around using a duplicate name, you have a choice among one of the following:

First Searches branch. Place the duplicated name in the First Searches branch. Below the name insert statements describing the different uses of the term with links to its different locations. Although this is the easiest choice for the description file maintainer, it cannot be used for names that are the same as tools. In index files the first tool or subsystem name must link directly to the file containing the tool/subsystem if the user is to be able to get a command description by typing the tool/subsystem name followed by the command name.

Combine descriptions. If the two descriptions are not conceptually much different they should be merged. The description branch should be placed in the location where it will most often be read, and links to it should be placed in the other location(s). This not only solves the duplicate name problem, but is useful in general to avoid duplication.

Reference second description. In the description of the first name of a duplicate name in a file, provide a reference (link or "See...") to the second description with the same name. Remember to use the right series of words when specifying the term.

Renaming. The only reason for using these methods is that the duplicated name is necessary for a good intuitive name. Otherwise, the second concept should go unnamed or be renamed, thereby eliminating any duplication. This can be done by hyphenating two words or adding a new term. Care should be taken when adding a new term that the description it names will be meaningful to the user who might type that term out of the blue. If there are other meanings for the new term, you are back to the original duplicate name problem.

When to Duplicate

Whenever a new statement name is created, its description must be examined from the point of view of a new user who happened to type the term hoping to get an understanding of its meaning. If the new statement name is being used in a very special case, another describing the more general case must be added in the appropriate branch.

Every statement name that has a possible ambiguous meaning to a naive or experienced user should provide a choice for or otherwise explain the ambiguous meaning. This is usually done by adding it to the First Searches branch and linking off to where the name normally occurs.

Duplicate Names Across Files

Some names will be duplicated across description files rather than in the same file. If your term means different things in different files, you may wish to place the duplicated name in the index file with links off to its definitions in each file.

Update Compact

The Help command is assured of finding the proper occurrence of a duplicate statement name in the description file only if one of the "choices" mentioned above was followed and the Base "Update File Compact" command is used after additional names have been added.

Comment Statements

There is a facility for making comments either visible or invisible to the user. Preceding a statement with a right square-bracket] means do not menu (number) this statement-- instead show all of its lines to the user of the Help command. Substructure under such a statement cannot normally be seen by the user. It is seen only if the substatement is named and the user happens to type the name.

Preceding a statement with a percent sign % means don´t show this statement to the user when he is using the Help command. This is useful for making comments to yourself or other description file builders. You can see the statement when you are not using the Help command. You can also place under an invisible comment named statements that the user can find only if he types the name. These are placed in a statement whose source is a statement beginning with a percent sign.

Style

Point of View and Audience

Use second person singular present tense. This familiar form provides directness and eliminates the gender problems and space consuming gymnastics necessary for third person. Write the description as if you are standing over the shoulder of the user and answering a question. Try to make all descriptions clear and straight-forward, avoiding jargon as much as possible. For basic terms and concepts gear your description to the naive user. For more advanced descriptions, you may want to assume more knowledge on the user's part.

Each description must make sense from five directions:
1) in menued sequence with its surrounding statements
2) typing its name out of the blue to get a definition of the term
3) typing its name as referenced from a "See"
3) from all the possible places that link to it
5) from the point of view of the person who just pushed the HELP button.

Description file builders tend to favor number 1 when in fact, because of the limitations of the Help accessing system, it is the least used. This is by far the most difficult, and least understood, area of description file development.

Definition

In general, the descriptions of every term should start with a glossary-type definition using as little jargon terminology as possible. Only in such cases where the definition is an incomplete sentence may the first word of the second line of a description be lower-case.

Runon Words

In many cases two words will be run together as one or hyphenated to maintain uniqueness, or because statement names may not contain spaces. For runon words that are statement names, place the more conventional spelling of the word(s) in the First Searches branch using a statement and substatement, linking to the description of the term--e.g.:

```
   1A run
      1A1 on:  runon words
          ##<runon>##
```

(This may not be practical if it creates a duplicate name.)

Capitalization

There are three kinds of capitalization: capitalized (initial upper), all caps, and all lowercase. Command words and names of things such as directories should be capitalized. All caps, as the only highlighting facility available online, should be used sparingly. It is reserved for acronyms, initials (such as prompts), and global variables or parse functions as specified in syntax notation. Words, phrases, or sentences beginning a description should be capitalized according to normal grammatical rules: only capitalize sentences. If in doubt, use lowercase. (Note that this file does not necessarily follow these rules since it is written for both online and offline use.)

REFERENCES

1. Andrews, Donald I., Beverly R. Boli, and Andrew A. Poggio. An Introduction to the Frontend. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. February 3, 1977. (28743,).

2. Andrews, Donald I., Lawrence L. Garlick, and Andrew A. Poggio. Frontend System Documentation. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. February 3, 1977. (28745,).

3. Shantz, R. E., and R. E. Milstein. The Foreman: Providing the Program Execution Environment for the National Software Works (Preliminary). Bolt, Beranek, and Newman, Boston, Mass. and Massachusetts Computer Associates, Wakefield, Mass. March 31, 1975.

4. White, James E. A High Level Framework for Network-Based Resource Sharing. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. December 23, 1975. (27197,)

5. White, James E. DPS-1Ø Version 2.5 Programmer´s Guide. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. December 23, 1975. (26271,)

6. Andrews, Donald I. Line Processor: A Device for Amplification of Display Terminal Capabilities for Text MAnipulation. AFIPS Proceedings, National Computer Conference. June 1974. pp. 257-265. (2Ø184,)

7. Hardy, Martin E. Micro Processor Technology in the Design of Terminal Systems. Proceedings of the IEEE COMPCON, 1974. (2Ø185,)