STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

FRONTEND SYSTEM DOCUMENTATION

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

January 5, 1977

ARC Catalogue Number 28745

TABLE OF CONTENTS

PREFACE 1
CLI OPERATION 2
The CML Grammar 2
CML Grammar Interpretation 3
CML COMPILER AND COMPACTOR DESCRIPTION 4
Introduction 4
The Compiler 4
The Compacter 4
CML VARIABLE TYPES 23
Introduction 23
CML Variable Structure 23
CML Types 23
VIRTUAL TERMINAL CONTROLLER DESCRIPTION 26
Introduction 26
VTC Design 26
VTC Implementation 28
Glossary 34
PROCESS COMMUNICATIONS INTERFACE (PCI) MODULE 37
Introduction 37
Important Data Elements 37
PCI Procedures 37

DATA REPRESENTATION INTERFACE 40

Introduction 40

Reading a PCP Data Structure 40

Writing a PCP Data Structure 44

USER PROFILE DATA STRUCTURE AND TOOL 47

Introduction 47

Current Capabilities

Envisioned Capabilities 47

User Profile Data Structure 47

GENERATING A NEW FRONTEND 52

Introduction 52

Compilation 52

Loading 52

Frontend Files 53

REFERENCES 56

PREFACE

This document, which reflects the current state of the NSW Frontend, provides information about the Frontend necessary to those working with it or building their own Frontend. The three classes of Frontend components and the extent to which this document describes them follows:

Modules. The programs with which the user interacts during command specification and which communicate with the tool: the Virtual Terminal Control (VTC), the Command Language Interpreter (CLI), and the Process Communication Interface (PCI). A description of CLI operation and of the PCI and VTC is provided. Also included is a discussion of data representation types.

Data Bases. The data bases and data structures associated with the user interface machinery: the grammar, User Profile and Help data bases, User Statistics, CML source programs, and Command Sequences. More information on the data bases may be found in A GUIDE TO THE CML AND CLI [1].

Auxiliary Tools. The auxiliary programs and tools that allow the user or tool builder/installer to create, examine, or manipulate the above data bases: the CML compiler, User Profile tool, Help tool, Statistics Analysis programs, and Command Sequence Processor. A detailed description of the CML compiler and compacter and CML variable types is provided here, along with a section on the current and future capabilities of the User Profile tool and its data structure.

The last section tells briefly how to create and load a Frontend.

CLI OPERATION

The CML Grammar

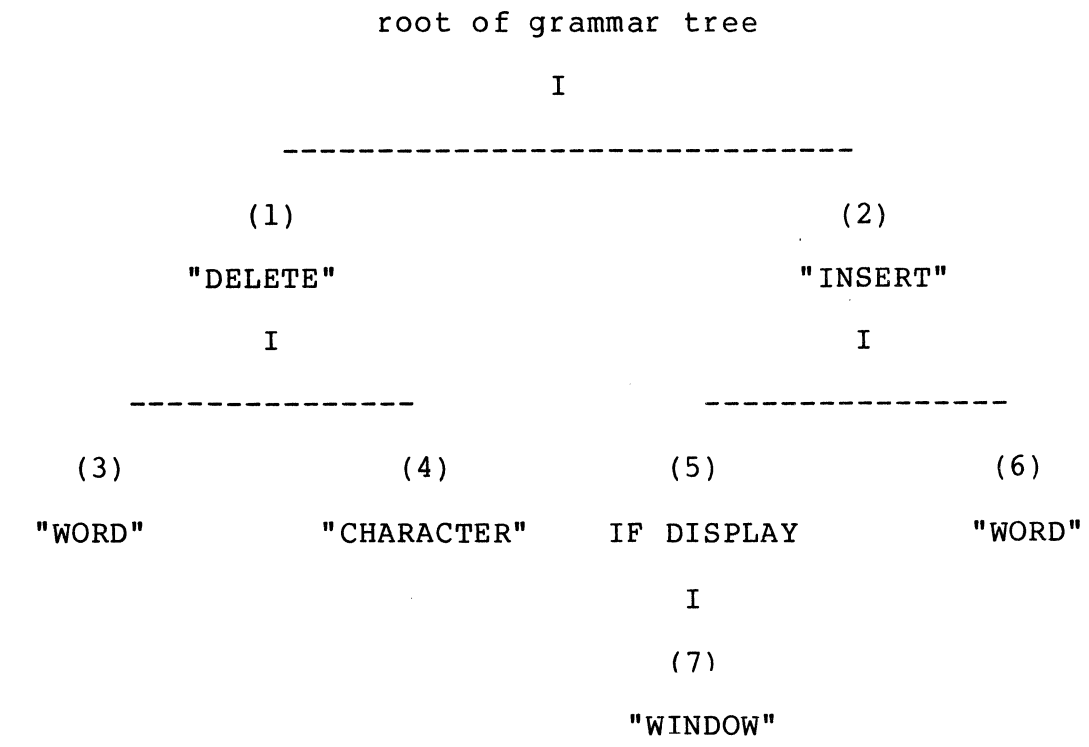
A CML grammar consists of a series of instructions and associated tables. The grammar instructions form a tree-structured program which the CLI interprets. It is this process of grammar interpretation that produces the high quality user interaction for which the CLI is so well known.

A CML grammar consisting of the two commands,

```
delete COMMAND = "DELETE" ( "WORD" / "CHARACTER" );

insert COMMAND = "INSERT" ( IF DISPLAY "WINDOW" / "WORD" );
```

when compiled produces the following (upside down) tree structure:



Each place in the tree where an instruction such as "WORD" exists is called a node. Each node has been numbered for

reference. Nodes 3, 4, 7, and 6 are called terminal nodes because there is nothing following them in the tree.

CML Grammar Interpretation

The CLI implements the command language described by a grammar by interpreting the instructions contained in the grammar. The CLI begins at the root of the grammar tree and simultaneously processes the various paths through the tree to terminal nodes. One such path is through nodes 1 and 3; another is through 2, 5, and 7. Completing the processing of a terminal node such as node 3 is synonymous with command completion and causes the CLI to begin processing again at the root of the grammar tree.

The CLI path processing may be directed by user input or results obtained from processing instructions. In the above example, the CLI would start by processing nodes 1 and 2 at the same time. Since both of these nodes are command words, user characters are required for the CLI to determine which path to take. Let us assume that the user is in expert recognition mode and that she types a "d". The CLI can then determine that node 1 is on the correct path and node 2 is not because the user must be typing the "DELETE" command word; as a result it discontinues processing the path through node 2 and proceeds down the path through node 1 to process nodes 3 and 4. The user types a "c". The CLI can then determine that node 4 is on the right path. Node 4 is a terminal node and so upon completion of the processing of node 4, the current command is complete and the CLI will begin processing again at the root.

If instead the user types an "i" as the first character of her command, the CLI will then process the nodes following node 2--nodes 5 and 6. Node 5 checks the value of the built-in variable DISPLAY which is TRUE if the user is at a display terminal. If DISPLAY is TRUE, the CLI is directed to continue down to node 7 and process it together with node 6; otherwise it will discontinue processing the path through node 5 and only process node 6. Let us assume that DISPLAY is TRUE and so nodes 7 and 6 are both processed. The user types a "w". The CLI cannot determine whether node 7 or node 6 is on the correct path because both are command words beginning with "w", and so it must get another character from the user to continue. Suppose this next character is "x". The CLI finds that neither node 7 nor node 6 is correct for this character and there are no other currently active paths. The CLI then assumes that the user typed a bad character, tells her so, and gets another character which hopefully will be "i" or "o", thus allowing the CLI to finish processing the command.

CML COMPILER AND COMPACTOR DESCRIPTION

Introduction

The CML compiler and CML compacter are used to transform the text of a CML grammar into a form executable by the CLI. The CML compiler takes as input a source file, either an NLS or text file, and produces as output a file containing an appropriate set of CML instructions. The CML compacter is a post-processer for the compiler which reduces the size of the compiler output and produces any modifications necessary for the grammar to run on its target machine.

The following discussion assumes that the reader is familiar with the CML and L10 languages.

The Compiler

The CML compiler is written in Tree Meta, a compiler-writing language, and runs on the PDP-10. It can be run in NLS taking as input an NLS file, or from the EXEC with a text file as input.

The format of the compiler output is a directed graph whose nodes are instructions, each of which occupies two 36-bit words. The links in the directed graph are implemented through two fields in each instruction--the alternative field and the successor field. The alternative field contains the address of the CML instruction to execute in parallel with this one, while the successor field contains the address of the next instruction to execute should this one succeed.

Another field in each instruction indicates the type of the instruction, such as "recognize a command word" or "call an execution function". Other fields contain information dependent on the instruction type. For example, the "recognize a command word" instruction has a field which contains a pointer to the command word string.

The compiler output is a relocatable file which must be link-loaded before being used.

The Compacter

Overview

The compacter is written in the L1Ø language and runs on a PDP-1Ø. The input to the compacter is the compiler output file; its output is a compacted grammar file. The compacter further processes the output of the compiler to reduce its size and put it into a form appropriate for the type of computer that it will be running on. As a result, there are two compacters, one for producing grammars to run on the PDP-1Ø and one for the PDP-11.

The output of the two compacters differs in only two ways. The first is the dispatch record at the beginning of the grammar. This record is specially formatted for the PDP-11 so that the same L1Ø RECORD definition can reference it on both the PDP-1Ø and PDP-11.

The second difference lies in the way strings and pointers to strings are handled. For the PDP-1Ø, characters are packed five to a word. String pointers point to the word previous to the string, which contains the maximum length of the string in the left half of the word and the current length of the string in the right half, i.e., M,,L. For the PDP-11, characters are packed two to a word, with the exception of the first word which contains only the first character of the string in the high byte. (String pointers point to this first word.) The current length of the string is in the word previous to where the string pointer points; the maximum length of the string is in the word previous to that. You may already have noticed that strings and string pointers in compacted grammars are implemented in a manner identical to that of the L1Ø and L1Ø11 languages.

All further discussion of the compacter applies to both the PDP-1Ø and PDP-11 compacters.

A compacter operates by loading the relocatable file produced by the compiler, together with any related parse function files. Thus, the grammar has all of its references resolved and may be put into a form that need not be link loaded. Upon completing the compaction processing, the newly-produced compacted grammar is written on a file.

Compacted Grammar Description

Overall Structure

A compacted grammar consists of two segments: a code segment and a data segment. The code segment contains the dispatch record, various tables used by the grammar, and the CML instructions. The data segment contains the grammar's variables and process records.

The Code Segment

The first item in the code segment is the dispatch record. It contains pointers to the tables in the segment, byte numbers of certain instructions in the grammar, and other information about the grammar. Pointers are relative to the beginning of the segment starting at 0; the byte numbers are relative to the beginning of the segment starting at 1; a 0 byte number indicates the absence of an instruction. The dispatch record has the following L10 RECORD definition:

```
(subr) RECORD % grammar dispatch record %

  subname[ADDRESS],     % pointer to subsystem name string %

  firstinst [ADDRESS],  % byte number of first instruction of
  commands %

  valcode[ADDRESS],     % validation code - 10 for the PDP-10,
  11 for the PDP-11 %

  hlprule[ADDRESS], % grammar help rule / 0 indicating none %

  initinst[ADDRESS],    % initialization rule / 0 %

  reeninst[ADDRESS],    % reentry rule / 0 %

  termrule[ADDRESS],    % termination rule / 0 %

  prsrec[ADDRESS],  % pointer to process records relative to
  beginning of DATA segment / 0 indicating none %

  kword[ADDRESS], % pointer to command word table %

  echoword[ADDRESS], % pointer to noise word string table %
```

```
execvector [ADDRESS], % pointer to execute byte number
table %

pfuncs [ADDRESS], % pointer to parse function address table
%

funcs [ADDRESS], % pointer to function record table %

gvstart [ADDRESS], % pointer to global variables relative
to beginning of DATA segment %

sharl [ADDRESS], % number of 32 word blocks in code segment
%

priv [ADDRESS], % not used %

privl [ADDRESS], % number of 32 word blocks in data segment
%

pfcsize[ 8 ] % number of 32 word blocks in parse function
code %,

pfdsize[ 8 ] % number of 32 word blocks in parse function
data %;
```

Following the dispatch record in the code segment are the CML instructions. If an instruction has an alternative instruction, it is always the next instruction. The successor of an instruction, if it has one, always follows the instruction, although it is not necessarily the next instruction. Instructions are coded into 8-bit bytes. There are byte numbers for certain of these instructions--e.g., the first instruction of the commands--in the dispatch record. A description of the instruction format is given below in "Compacted Grammar Instruction Format".

The tables for the grammar follow the instructions. With one exception, an entry in any table consists of a pointer relative to the beginning of the code segment. The single exception is the execute byte number table (pointed to by the 'execvector' field) whose entries consist of the byte number of an instruction relative to the beginning of the code segment. All table entries occupy a full computer word.

Following the tables are various constant data elements used by the grammar, e.g., command word strings. Command words defined as selectors are slightly different from other command words, in that the three words previous to the command word string are indices into the parse function table. These indices indicate

the selection parse functions to be used to gather a selection by pointing, typing in, and typing the address. The order of the three words is:

point selection parse function index

type in selection parse function index

type address selection parse function index

command word string

The selection parse function may be built into the CLI, as with a TEXT selector for example, or may be written by a CML programmer. A zero index indicates that selection by the corresponding method is undefined. For example, a zero for the point selection parse function index means that the selection cannot be pointed to.

The Data Segment

The data segment of a compacted grammar contains those elements which may change during the use of the grammar; this prevents it from being shared among multiple users of the grammar as is the code segment.

The first elements in the data segment are the grammar variables. During execution of the grammar, these elements will contain pointers to the actual values of the grammar's variables. Typically, these point into the free space area of the CLI. The variables are divided into two groups: local variables followed by global variables.

Following the variables in the data segment are the process records. These are four word records, one for each backend process that the grammar might interact with, as given in the grammar source. The format of these records is dependent on the interprocess communication protocol being used by the FE.

Compacted Grammar Instruction Format

Each CML instruction consists of one or more contiguous 8-bit bytes. The first byte of an instruction always indicates its type. Following the first byte there may optionally be one or more bytes of fields related to the instruction type. In addition, there may optionally be one or two bytes of successor field.

The first byte of an instruction has three fields, defined as an L1Ø RECORD as follows:

```
(instrec) RECORD opcode[5], altsuc[2], llcmd[1];
```

The ´opcode´ field indicates the instruction type, e.g. ´recognize a command word´ or ´call an execution function´. The ´llcmd´ field is used by different types of instructions in different ways. For example, the ´recognize a command word´ instruction uses it to indicate whether its command word is first level or not.

The ´altsuc´ field, together with the optional successor field, provides information about the instruction´s alternative and successor instructions. It may assume the following values, which are defined as external constants in the CLI:

notlast--The instruction is not the last alternative and its alternative instruction is the next instruction following it. A successor field indicates the successor´s location.

lastnone--The instruction is the last alternative, i.e., it has no alternative and no successor.

lastfield--The instruction is the last alternative and its successor is indicated by its successor field.

lastnext--The instruction is the last alternative and its successor is the next instruction.

A successor field may be used to indicate the location of an instruction´s successor. Whether or not an instruction has a successor field is indicated by the ´altsuc´ field in the first byte. A successor field contains the displacement of the successor instruction relative to the last byte of the successor field. A displacement of 3 would mean that the successor instruction begins at the third byte following the last byte of the successor field. A zero successor field indicates that the instruction has no successor.

A successor field may be one or two bytes long; its first byte has two fields with the following L1Ø RECORD definition:

```
(sucrec) RECORD sucadd[7], long[1];
```

If the 'long' field equals 0, the value of the successor field is simply the value of the 'sucadd' field. If the 'long' field equals 1, there are two bytes of successor field. The value of the field is

```
value = ( sucadd field * 256 ) + second byte of field
```

Other fields in instructions are often indices into one of the grammar tables. Table indices always start at 0. They are used to compute the absolute addresses of various grammar elements. For example, to compute the absolute address of a command word in the grammar given its index 'ind', the following steps are taken:

1. Compute the address of the beginning of the command word table 'comwordtab', given the address of the beginning of the code segment 'codseg', and using the 'kword' field of the grammar dispatch record:

```
comwordtab _ codseg + [codseg].kword;
```

2. Compute the absolute address of the command word 'comwordaddr':

```
comwordaddr _ codseg + [comwordtab][ind];
```

Some fields in instructions are variable designators. These are used to locate variables referenced by the grammar. An 8-bit byte used as a variable designator is formatted according to the L10 RECORD declaration

```
(addrrec) RECORD varind[6], vartype[ dummy[1], notlocal[1] ];
```

If the 'notlocal' field equals 0, the variable is and the entire byte is an index whose base is the beginning of the grammar data segment. If the 'notlocal' field equals 1, the 'vartype' field is used to determine whether the variable is

global--meaning the 'varind' field is an index into the data segment with base indicated by the dispatch record field 'gvstart', or

bltin--meaning the variable is built in to the CLI and 'varind' is an index into the CLI's array of built-in variables.

Instruction Types

The instruction types--defined as external constants in the CLI--and the format of their bytes are described below. All instructions use the 'opcode' and 'altsuc' fields. These descriptions do not include the successor field, which may or may not be present in a particular instruction. The value of the instruction types in octal is given in parenthesis.

abortop (0). Show the user the contents of the CLI accumulator and abort the command.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

keyop (1). Recognize a command word. The command word may be a literal, or a variable containing a string, or a list of strings.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

llcmd--this field equal to 1 implies command word is first level.

second byte: This entire byte is interpreted differently depending on the 'kwvar' field in the third byte and is either:

an index into the command word table if the command word is a literal, or

a variable designator if the command word is a variable.

third byte: (valrec) RECORD numofargs[tstrel[3], tstint[1], tstnot[1], filler[1]], hashelprule[1], kwvar[1];

kwvar--this field equal to 0 implies that the second byte contains an index into the command word table and the entire third byte is the integer token associated with the command word. This field equal to 1 implies that the second byte contains a variable designator and the remainder of the third byte should be ignored.

No other fields are used.

confirm (2). Get a confirmation from the user, e.g., by the user typing a COMMAND ACCEPT.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

ssel, dsel, lsel (3, 4, 5). Get a source, destination, or literal selection from the user.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

pusharg (6). Push the CML value in the accumulator onto the argument stack. This is the method by which arguments are gathered for parse or execution function calls.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

answer (7). Get an answer from the user, e.g., by the user typing "y" for "yes".

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

option (10B). Get an OPTION character from the user.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

pfcllop (11B). Use a parse function.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is an index into the parse function address table.

third byte: (valrec) RECORD numofargs[tstrel[3], tstint[1], tstnot[1], filler[1]], hashelprule[1], kwvar[1];

numofargs--the number of arguments that the parse function is to be OPENPORTed with.

No other fields are used.

execute (12B). Execute a series of CML elements.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is an index into the execute byte number table.

call (13B). Call an execution function.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

llcmd--this field equal to 1 implies that the function is to be called in "out of line" mode.

second byte: entire byte is an index into the function record table.

third byte: (valrec) RECORD numofargs[tstrel[3], tstint[1], tstnot[1], filler[1]], hashelprule[1], kwvar[1];

numofargs--the number of arguments that the function is to be called with.

No other fields are used.

fbclear (14B). Clear the feedback window.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

echo (15B). Put a noise word string in the feedback window.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is an index into the noise word string table.

recho (16B). Remove the last item from the feedback window and put in a noise word string.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is an index into the noise word string table.

storeop (17B). Store the CML value of the accumulator into a CML variable.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is a variable designator.

load (20B). Load the accumulator with the value of a CML variable.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is a variable designator.

enter (21B). Enter a number into the accumulator.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is the number to enter into the accumulator.

test (22B). Test the value of the accumulator.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is interpreted differently depending on the 'tstint' field in the third byte. The second byte is either:

an integer to compare the accumulator with, or

a variable designator to compare the accumulator with.

third byte: (valrec) RECORD numofargs[tstrel[3], tstint[1], tstnot[1], filler[1]], hashelprule[1], kwvar[1];

tstrel--the relation which is being tested for. The following, which are declared in the CLI, are possible values:

equal

greater

less

grequal

lessequal

tstint--this field equal to 1 implies the test is an integer test and the second byte contains the integer. This field equal to Ø implies a test against a variable and the second byte contains a variable designator.

tstnot--this field equal to 1 implies that the result of the test should be complemented.

showop (23B). Show the contents of the accumulator to the user.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

entercw (24B). Enter into the accumulator a command word.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: the entire byte is an index into the command word table.

third byte: the entire byte is the integer token associated with the command word.

enternull (25B). Enter into the accumulator a NULL.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

entertrue (26B). Enter into the accumulator a TRUE.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

enterfalse (27B). Enter into the accumulator a FALSE.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

resume (30B). Resume a help call.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

append (31B). Append the value of the accumulator to a CML variable.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is a variable designator.

testtrue (32B). Test to see if the accumulator is TRUE.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

llcmd--this field equal to 1 implies that the result of the test should be complemented.

testnull (33B). Test to see if the accumulator is NULL or FALSE.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

llcmd--this field equal to 1 implies that the result of the test should be complemented.

helpcall (34B). Call an execution function that has specified a help rule.

first byte: (instrec) RECORD opcode[5], altsuc[2], llcmd[1];

No other fields are used.

second byte: entire byte is an index into the function record table.

third byte: (valrec) RECORD numofargs[tstrel[3], tstint[1], tstnot[1], filler[1]], hashelprule[1], kwvar[1];

numofargs--the number of arguments that the function is to be called with.

No other fields are used.

fourth byte: the entire byte is an index into the execute byte number table indicating the first instruction of the help rule.

Example

The following provides a comparison between a CML grammar source and the corresponding compacter output. The source is: FILE cmlexp %<nsw-sources,cgcml,>
<poggio,cmlexp.cml,>%

```
% Declarations %

  DECLARE COMMAND WORD

    "SHOW",

    "CML" = 1,

    "CLI" = 2;

  DECLARE VARIABLE var;

  DECLARE FUNCTION doexample;

SUBSYSTEM cmlexp KEYWORD "EXAMPLE"

  exp COMMAND = "SHOW" var _ ( "CML" / "CLI"!L2! )
  <"example"> doexample( var );

  END.

FINISH
```

The compacter output is shown in octal 8-bit bytes.
Instruction boundaries are indicated by dashes.

```
341
```

Recognize first level command word "SHOW". This instruction has no alternative and its successor is the next instruction.

```
2
```

"SHOW" is the third entry in the command word table.

```
0
```

Its integer token is 0 because none was declared.

201

Recognize first level command word "CML". This instruction has an alternative which is the next instruction. It has a successor field which indicates the location of the successor.

1

"CML" is the second entry in the command word table.

1

Its integer token is 1.

4

This is the successor field. The 'long' field of the byte is 0, indicating that the successor field is only 1 byte long. The successor field has a value of 4, indicating that the successor begins at the fourth byte following this one. This is the 157 byte, the first byte of the store instruction.

241

Recognize second level command word "CLI". This instruction has no alternative and its successor is next.

0

"CLI" is the first entry in the command word table.

2

Its integer token is 2.

157

Store the contents of the accumulator into the variable ´var´. This instruction has no alternative and its successor is next.

Ø

The ´notlocal´ field equals Ø implying that this is a local variable. It is the first local variable in the data segment.

155

Show the noise word "example" to the user. This instruction has no alternative and its successor is next.

Ø

The noise word "example" is the first entry in the noise word table.

16Ø

Load the accumulator with the contents of the variable ´var´. This instruction has no alternative and its successor is next.

Ø

The ´notlocal´ field equals Ø implying, that this is a local variable. It is the first local variable in the data segment.

146

Push the contents of the accumulator onto the argument stack. This instruction has no alternative and its successor is next.

53

Call the execution function "doexample". This instruction has no alternative and no successor.

Ø

The execution function "doexample" is the first entry in the function record table.

1

The ´numofargs´ field equals 1 indicating that one argument should be popped off the argument stack and passed to the execution function.

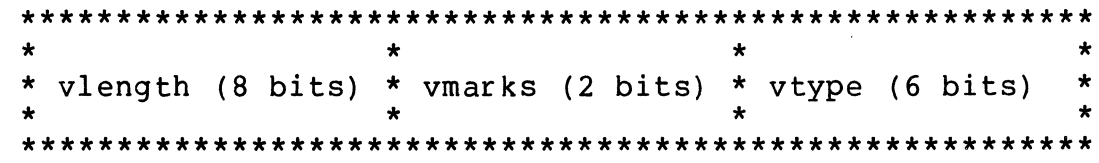
CML VARIABLE TYPES

Introduction

This section describes the Command Meta Language (CML) variable types and their translation into PCP types. It is intended for CML parse function writers but may be of interest to anyone familar with CML.

CML Variable Structure

CML variables all contain a pointer to a block of one or more contiguous words of memory. The first word (word 0) of the block is always a header with the following fields right adjusted in word:



As their names suggest, the vlength field indicates the variable length and the vtype field the variable type. The vmarks field indicates how many marks were made on the display during the creation of the variable; its value, typically zero, will depend on how the variable was created.

The header fields are defined in L10 as:

```
(var) RECORD  %header of a variable%

vtype[6], vmarks[2], vlength[8];
```

CML Types

STRING

word 0: vtype = strtype (= 1), vlength = 3

word 1: integer token associated with string or 0 if none

word 2: address of L10 string

COMMAND WORD

word 0: vtype = cwtype (= 2), vlength = 3

word 1: integer token associated with string or 0 if none

word 2: address of L10 string

The distinction between a command type and a string type is that a command word string may have been defined to be a SELECTOR. In this case the three words previous to the string may point to selection functions for pointing, addressing, and typing in. This fact may be generally ignored by the CML programmer.

INTEGER

word 0: vtype = integer (= 3), vlength = 2

word 1: the integer

POINT

word 0: vtype = pointtype (= 4), vlength = 5

word 1: the window identifier

word 2: the string identifier

word 3: the line segment identifier

word 4: the character position

ADDRESS

word 0: vtype = addrtype (= 5), vlength = 4

word 1: integer token associated with string or 0 if none

word 2: address of L10 string

word 3: window identifier

NULL

word 0: vtype = nulltype (= 6), vlength = 1

LIST

word 0: vtype = listtype (= 7), vlength =2 + number of elements in list

word 1: number of elements in the list

word 2 to word N: addresses of elements in the list (elements may be of any type)

TRUE

word 0: vtype = truetype (= 8), vlength = 1

FALSE

word 0: vtype = falsetype (= 9), vlength = 1

BLOCK

word 0: vtype = blocktype (= 10), vlength = number of bits in block

word 1 to word N: the bits (32 per 36-bit word left-adjusted)

WINDOW

word 0: vtype = windtype (= 11), vlength = 2

word 1: the window identifier

ADDRESS EXPRESSION

word 0: vtype = adexptype (= 12), vlength = 2 + number of elements

word 1: number of elements in the address expression

word 2 to word N: addresses of elements in the address expression

VIRTUAL TERMINAL CONTROLLER DESCRIPTION

Introduction

The Virtual Terminal Controller (VTC) module of the Frontend presents a virtual terminal interface to the tools and the Frontend itself. It contains the procedures and data to map the virtual operations into the actual operations necessary for communication with the connected terminal device.

The VTC defines three classes of terminals: (1) half duplex (possibly line at a time) typewriters, (2) full duplex typewriters, and (3) alphanumeric display terminals, perhaps with pointing devices, etc. A set of operations are defined for each class of terminal. Because operations are the same for classes 1 and 2, tools address themselves to two virtual terminal types: typewriter and display. More advanced graphics terminals fall in a fourth class, but the operations for this class are not yet specified.

The VTC functions as a service module in the FE. When a call is made on the VTC, it performs some specified function and returns. The VTC is accessed through one of two entry point procedures, which in turn call on the other VTC procedures. One entry procedure is used only by the Frontend to manipulate the terminal in some tool-independent fashion; the other is used when a tool explicitly calls on the VTC.

VTC Design

This section describes VTC capabilities and the design approach to the VTC. A glossary of terms used in this description may be found at the end of this chapter.

Capabilities

For typewriter terminals, VTC capabilities consist of setting the terminal device type, writing strings on the terminal, and controlling the carriage position.

A rich set of primitives exists for the display class of terminals, many of which rely on the concept of a display "window". Several windows are predefined by the Frontend and created by the FE through VTC primitives:

A TTY-simulation window for status or error messages.

A command feedback window.

One or more tool windows.

A special small window for tool mode information.

These windows are created by the Frontend proper by way of VTC primitives.

The screen may contain adjacent and overlapping windows, much the way a person views several pieces of paper on his desk. Each window has an associated priority to determine which window is visible when the windows overlap.

Tools are given custody of a window, called a primary window, when they are first started. The tool is then free to write and delete strings in the window, clear the window, and create more windows within it. Typically the primary window is nearly all of the screen (e.g., 20 out of 24 full lines).

A user can maintain separate primary windows for many tools concurrently by instructing the Frontend to divide an existing window. Primary windows may not overlap and can only be reconfigured and written by the owning tool. Within the boundaries of a primary window, a tool may create overlapping windows.

When using a display terminal, the user can select any text visible to him instead of typing it on the keyboard. This, combined with the ability to run severals tools in different windows concurrently, gives the user a helpful cross-tool facility.

When a tool is terminated, via a call to 'toolrst', all windows assigned during the tool's use are released. The data structure tool list (tlist) is used to find the windows associated with the tool, given the tool code.

Design Approach

The VTC is comprised of a collection of service procedures and a data base. The data base contains structural and textual information about the screen contents which the procedures manipulate in useful ways.

The data base consists of a minimum of "global" information that is always present, for the most part runtime-allocated blocks of data, of variable size, that describe the structure

and contents of the windows. This frees memory when not in use, and makes possible a more efficient use of memory by sharing the allocation pool among many Frontend processes.

The most important performance criterion for the VTC is response time, i.e., the speed of display manipulation. Hence a great deal of care is taken to make display manipulation efficient. This even manifests itself to the tool in the form of the "batch-commands" procedure, which will perform many display operations at one time and more efficiently than if done by many calls.

VTC Implementation

This section briefly describes the nature of the VTC interfaces, data structures, storage management techniques, and error handling strategy. It provides references that will be useful in locating functional areas within the source code.

Display Package Interfaces

The VTC module has three logical interfaces, the external tool interface, the internal CLI interface, and the terminal device interface.

A call generated from the CLI is of the form

```
DPYCALL(name, n, a1, ...an)
```

where name (an integer) is an internal VTC procedure number, and n is the number of arguments a1 through an.

The procedure DPYCALL calls the specified internal display procedure (many of which correspond one-to-one to the external procedures, but with the arguments in a different form). The procedure list is in array IDPYTAB. The association between the procedure name and its number is shown.

(scrollwindow = 1)--scroll a window.

(getdstr = 2)--get display string from display structure.

(toolset = 3)--set a window to be given tool's primary window.

(toolrst = 4)--remove tool from tool list.

(wrtlsg = 5)--write line segment.

(delstr = 6)--delete a string.

(dellsg = 7)--delete line segment.

(rpllsg = 8)--replace line segment.

(ppointsel = 9)--point selection routine.

(xywindow = 10)--given x, y real coordinates, return window-id.

(wrtstr = 11)--write string.

(crewind = 12)--create window.

(intseqw = 13)--initialize sequential window.

(setdftty = 14)--set default TTY window.

(swatt = 15)--set window attributes.

(setsatt = 16)--set string attributes.

(inwind = 17)--determine if a "window-relative" point is in a window.

(setlatt = 18)--set linseg attributes.

(delwind = 19)--delete window given window-id.

(delsubs = 20)--delete all sub-windows given window-id.

(clrwind = 21)--clear window, given window-id.

(markfcn = 22)--mark screen to show or remove a selection.

(wrtliteral = 23)--write literal string in a window.

(wrtpart = 24)--write partial string, subordinate to wrtliteral.

(finish = 25)--finish display manipulation sequence.

Calls to the display package from tools are external procedure calls as defined in Appendix 4 of A GUIDE TO THE CML AND CLI. The CLI transforms that into a call of the form

EDPYDSP(iport, oport)

where the two arguments are port-identifiers for source and sink of the current data representation (arguments and results). The display package procedures read their arguments and generat theri results using the two port-identifiers.

VTC Routines Available through the CLI

The procedure EDPYDSP dispatches external calls to the correct procedure, passing two coroutine port-ids as arguments, one for the coroutine that supports reading argument lists (rlist) and one for writing result lists (wlist). Each procedure reads the PCPB8 argument list, performs some function, and then writes a PCPB8 result list.

The procedure names below are those found in the source code of the VTC module.

(ecrewind)--create window.

(eclrwind)--clear window.

(escroll)--scroll window.

(ewrtlsg)--write line-segment.

(edelwind)--delete window.

(ewrtstr)--write string.

(edellsg)--delete line-segment.

(edelstr)--delete string.

(erpllsg)--replace line-segment.

(ewrtlit)--write literal into window.

(esetsatt)--set string attributes.

(esetlatt)--set line-segment attributes.

(eposstr)--reposition string.

(eposlsg)--reposition line-segment.

(erplstr)--replace string.

(ebatch)--batch commands processor.

(egetwindows)--get window information.

(etrack)--turn cursor tracking on/off for graphics terminal.

The VTC external procedures are callable through a set of externally callable routines in the FE. The list of user callable procedures with their arguments and results are described in A GUIDE TO THE CML AND CLI, Appendix 4, "Externally Callable Procedures in the Frontend".

Terminal Device Interface

Keyboard and Line Processor input is performed by the procedure 'dinptc', which makes use of the following procedures:

(dinptc)--single character input routine.

(lpgetchar)--line processor GETCHAR routine.

(angetchar)--getchar for alphanumeric displays.

(ancnv)--alphanumeric display convert typein to coordinates.

(dinbc)--read a big character.

(dinbcl)--read mouse button changes.

Display and Line Processor manipulation is performed by the following procedures. They are called by various VTC prodecures to manipulate the terminal device.

(andspstr)--display string on alphanumeric display.

(pad)--pad with given number of null characters.

(anreset)--reset alphanumeric display.

(position)--position cursor - alphanumeric displays.

(trnslz)--translate coords for alphanumeric display.

(cscreen)--clear screen - alphanumeric displays.

(track)--resume tracking - LP displays.

(rtrack)--resume tracking - LP displays.

(cline)--write blanks - alphanumeric displays.

(dline)--delete line - alphanumeric displays.

(inline)--insert line - alphanumeric displays.

(standout)--Send stand out command - alphanumeric displays.

(endstandout)--Send end stand out command - alphanumeric displays.

(lpttywindow)--specify default tty - LP terminals.

(realcds)--convert relative coords to real screen coords.

(lsgtrncate)--line segment truncate function.

(sndlsg)--send line segment to terminal.

(dpysout)--display string output with control chars.

Special VTC Data Elements

The primary data elements are summarized here and described in detail in the code file. Below is a list of important data elements, with their corresponding names as found in the source code. In some cases, further description can be found in the glossary at the end of this chapter.

User information

(userinfo) RECORD--Each userinfo record contains informtion that is referenced during a terminal session. It contains terminal specific information, the tool-list address, and the window-list address.

Tool-list

(tlist) RECORD--A list of tool code and primary window-id pairs.

Window table

(wtab) RECORD--Each entry contains window size and location fields, priority, type, attribute list address, parent window-id, and bookkeeping for the contents of the window.

Attribute Word

(atts) RECORD--Each word contains attributes for linsegs or groups of linsegs (strings or windows). The attributes include visibility and high-lighting, the type of designation (text string or coordinates), and selector code.

String-list

(slist) RECORD--Each element contains a string attribute word, a linseg-list element address, and coordinates for the origin of the string.

Linseg-list

(lslist) RECORD--Each element contains a linseg attribute word, the address of the text string, and the coordinates of the origin of the linseg origin.

Garbage list

(garbgr) RECORD--each element contains a window-id and coordinates and character counts suitable for clearing lensegs from the screen.

Delayed write list

(dlyrec) RECORD--each element contains window, string, and linseg-id information for deferred updating of the screen.

Mark block list

(markr) RECORD--each element contains mark type, window-id, string-id, a pair of linseg-ids, and a pair of character counts used to record a marked (bugged) entity.

Storage Management

Storage management is handled by a standard package of routines that is used by most L1Ø modules. Interface to the package is provided by the procedures below. They assume that a storage management zone "dpyzone" has been initialized through a call to "makezone", a procedure in the storage management package.

(getdpy)--get block of storage from dpyzone.

(fredpy)--free block of storage from dpyzone.

Error Handling

Errors are handled via the L1Ø signalling facility. An abort may occur in any procedure and may be acted upon in any active catchphrase encountered up the thread of control. Aborts are generally ignored until they reach the top level dispatching routine (dpycall or edpydsp), at which time (finish) is called for cleanup.

Glossary

big character

An element of the Line Processor protocol--a sequence of characters in the terminal stream that begins with <ESC>. Used in conjunction with Line Processor terminals to send pointing coordinates and special characters.

coordinates

The horizonal (x) and vertical (y) displacements from the origin (lower left corner) of a window.

delayed writes

The method used to optimize display manipulation sequencing by doing all "writes" last. The writes are recorded in a list pointed to by dlyblk. Delayw(windtab, str, lsg) will record entries and wrtdely(TRUE) will actually do the writes. Wrtdely(FALSE) will delete the records without writing.

garbage list

A linked set of garbage blocks with "sizg" garbage elements in each. Each garbage element indicates a string of garbage that is on the screen and must be cleared sometime. Each element looks like "garbgr". Global "garbg" contains the address of the first block. Word zero of each block contains the address of the next (or zero if the last block).

linseg-id

Identifier for a line segment (linseg) in a specified string.

linseg-list

A list of linseg element records, one for each linseg in a given string. Each element points to a text string.

mark block

An allocated block which defines a selection mark on the screen: "markit" creates them and "popmark" uses them to remove the marks. They are then linked in reverse order through userinfo.mark (i.e., the most recent mark is first in the link). See userinfo and markr.

selector code

An 8-bit number that indicates the selectivity of strings on the screen. Three values are given semantics by the Frontend:

zero and one: Only selectable as literal (LSEL).

two: NEVER selectable as literal (LSEL) except across tools. (SSEL and DSEL okay).

greater than two: Semantics given by tool. All selections okay.

When a selection is made, a selector code argument is given to "ppointsel". Normally it looks only at strings with an exact match, except:

if arg <= 1: Anything is elegible.

if selcd = 2: Case is checked by parsefunction.

(Using arg = 2 is not normally done.)

string-id

An identifier for a string in a specified window.

string-list

A list of string element records. There is one element for each string in the specified window. Each string element record (slist) is referenced by a string-id and contains a pointer to either a linseg-list or a text string.

tool code

A WORD that uniquely identifies a tool.

tool list

A list that contains, for each tool for a given job, the pair tool code and window-id (window-id of that tool's primary window).

window

A rectangular area of the display screen.

window priority

An integer used to determine which of the overlapping windows will show. A "privis bit" is set for each window that indicates if the window is visible due to priority.

window-id

An integer that designates display window or other channel.

window-list

Contains the window-table-address for each window. Indexed by window-id.

window-table

Contains all the information pertaining to the given window, including the string-list address.

PROCESS COMMUNICATIONS INTERFACE (PCI) MODULE

Introduction

The Process Communications Interface Module (PCI) interfaces the Frontend to the communications media, and hence to other processes such as tools and the Works Manager. As described in AN INTRODUCTION TO THE FRONTEND [2], there is a PCI for each communication medium, each PCI supplying the same interface to the Frontend. The primary functions of the PCI are to provide a way for the Frontend to call remote processes, and for those remote processes to call Frontend (externally callable) functions and to allow for character-oriented communication.

Important Data Elements

The primary data element of the PCI is the process record ´processr´, which contains information necessary to communicate with a remote process. The actual information may differ depending on the communication medium, but it would typically include the process name and other identifiers for the process or connections to it. The process communication buffers record ´pcomrec´ is another important data element, containing pointers and addresses of send and receive buffers.

Character oriented communication is implemented as a Telnet network connection between the Frontend and the other (tool) process. A data element, the Telnet control block (TCB), is used to maintain each such connection the Frontend establishes.

PCI Procedures

The following procedures are used by the Frontend.

ipcinit (->)

This procedure has no arguments and no results. It is called at initialization time so that the PCI module may initialize itself.

ipcnewgram (instance REF ->)

The single argument ´instance´ is a grammar instance name. The procedure initalizes the process record for the remote process which that grammar will make calls upon. It is called each time a new grammar instance is created.

ipcendgram (instance REF ->)

This procedure is analogous to 'ipcnewgram' and is called whenever a grammar instance is no longer to be used by the FE.

ipccall (fn REF, outofline ->)

The procedure 'ipccall' performs the call on a remote process. The first argument is a CML function name identifier. The boolean 'outofline' indicates whether the PCI is to wait for the remote reply or return as soon as possible, processing the remote reply at a later time (FALSE implies waiting). This procedure decodes the parameters in the function name block, sets up the data to be communicated, and initiates the transmission.

ipcnetrec (echo, netinjfn, netoutjfn -> telcb)

This procedure is called to set up a character oriented (Telnet) connection to a tool. The call is made after the connections are established, but before they are used in any way.

The argument 'echo' is TRUE only if input echoing is to be done by the Frontend. (Normally echoing is done by the tool.) The arguments 'netinjfn' and 'netoutjfn' are handles on the 'input' (with respect to the Frontend) and the 'output' connections of the Telnet pair. The single result is the address of the Telnet control block for this connection pair. The procedure calling 'ipcnetrec' must remember the TCB address and delete the TCB when it closes the connection pair. This is usually done by a parsefunction, such as 'fetermtelnet'.

This procedure creates and initializes the Telnet control block, sending an initial Telnet option negotiation string to the server Telnet at the tool end. In the TENEX implementation, it creates a sub-fork to read characters from the connection and interrupt the main Frontend fork when characters are available. The main fork interrupt routine 'xtelnetpsi' (in PCI) then disposes of the characters as appropriate: it will reply to Telnet option negotiation by either outputting characters to the terminal or writing them in the proper window of the display screen. All the information needed by 'xtelnetpsi' resides in the Telnet control block for the connection in question. When there are several such connections, the control blocks are linked together and 'xtelnetpsi' handles each of them in turn.

The following PCI procedure is called when a remote call is made on the Frontend.

docall (inbuf REF, outbuf REF, outbufsiz -> reslen)

The procedure 'docall' performs the calls on externally callable Frontend procedures on behalf of remote processes. Its first argument is the address of the input buffer, which must contain a message-oriented procedure invocation of the form defined in Appendix 1 of A GUIDE TO THE CLI AND CML.

The procedure 'docall' removes the parameters from the top-level list (e.g., message type, procedure name), initializes the output buffer 'outbuf' for the results (if necessary), and calls the designated procedure.

To read the data types and values in the message, 'docall' uses the Data Representation Interface routines. It also passes on the port identifiers for the coroutines to the externally callable procedure. That is, every Frontend externally callable procedure is called with two arguments: the port identifiers of coroutines to read and write data structures, respectively. At the time of the call, the position within the 'input' data structure is such that the first element read is the first argument for the external call; likewise, the first element written in the output structure will be the first result, and so forth. The externally callable procedures are responsible for correctly reading their arguments using the port identifiers provided and building any result structures.

The following are externally callable procedures, described in Appendix 4 of A GUIDE TO THE CML AND CLI. They reside in the PCI module because most of the functions they perform involve the communications media.

feopenconn--whose internal name is eopenconn.

fecloseconn--whose internal name is ecloseconn.

fetermtool--whose internal name is etermtool.

DATA REPRESENTATION INTERFACE

Introduction

This section describes the L1Ø coroutines used to read/write PCP data structures. PCPB8 is described in A GUIDE TO THE CML AND CLI, Appendix 3, "Frontend Data Representatin for Message Communication". Since PCP data structures are sequential (i.e., there are no links), it is necessary to keep track of the current position in the structure while it is being encoded or decoded. L1Ø coroutines can perform the task of holding the current position, and thus are well suited to the encoding and decoding of PCP data structures.

Two things should be noted before we continue:

These coroutines assume the PCPB8 type PAD.

The coroutines are generally useful, and compatible on a PDP-1Ø (PCPB36 or B8) and PDP-11 (PCPB8).

Reading a PCP Data Structure

RLIST (adr REF, zone -> [iport])

To read a data structure, openport on rlist, providing the address of the PCP data structure (first word), and a free storage zone. The returned port ID will be used subsequently to read elements from the data structure, as described below:

rtype _ PCALL [iport] (type, length, dest: value, ptr)

´rtype´ is the actual element type.

´type´ designates the expected type of the element(s) to be read or a special operator.

´length´ is a count of the number of elements of type ´type´ to read into array ´dest´.

´dest´ is the address of an array to store element values in.

´value´ is the element value or a pointer.

´ptr´ is an address into the data structure that can be

used to reset your position to this point in the data structure.

The following are typical uses of one rlist PCALL:

rtype _ PCALL [iport] (Ø: value)

Will read one element of ANY type. dest and length need not be specified here when type is zero.

PCALL [iport] (pcpindex, $array, 5);

Will read 5 indexes and store the values in an array starting at 'array'.

PCALL [iport] (pcplist: listlen);

Will read one element, which must be a list. An abort will occur if it is not a list (err is called). The list length will be stored in listlen. Subsequent PCALLs will obtain the list elements.

In the above examples the returned 'ptr' was ignored. It could also have been stored.

These types are possible (see A GUIDE TO THE CML AND CLI, Appendix 4, for PCP type values):

type = pcpany (=Ø):

Any PCP data type except PAD is returned. The dest and length parameters, if present, are ignored.

type = pcplist

Here 'dest' is ignored. An abort is generated if the element is not a list, or if it does not have 'length' elements. The value returned is the number of elements in the list. Subsequent PCALLs will read the list elements. There is no indication of when the end of the list is reached; the element following the list will be returned after the last list element is returned.

type = pcpboolean, pcpindex, pcpempty

The type must match the element(s) being read or an abort will be generated. The value returned is the

value of the element. Zero is returned for an empty element.

type = pcpinteger

It is tricky to enable similar implementations on the PDP-10 and PDP-11. If the element is being returned as a PCALL result, the value will be the address of the integer (32 bits), which must be moved before the next PCALL. If the element is being stored in an array, 32 bits will be stored at the given location. This is a word on the PDP-10 and two words on the PDP-11, of course.

type = pcpcharstr

The character string is moved to the free storage zone and the address of the string is returned. On the PDP-11, if the zone is 'dpyzone', the string is not an a-string but is compacted; there are no M and L words and the length is in character zero.

type = pcppad

Pads are ignored and will never be returned; this will always fail.

type = rlistignore(100)

This will cause the next 'length' things to be ignored. They may be lists. This PCALL returns after advancing through the data structure. The type and value results are unspecified, but 'ptr' is correct.

type = rlistreset(103)

This will re-establish the position in the data structure to that given by 'length'. It must be a pointer obtained from another rlist PCALL (third result). The type and value results are unspecified, but 'ptr' is correct. The next PCALL will return the element following the one that was returned when the pointer was obtained.

type = rlistzone(101)

This will set the zone that rlist uses for storing pcpcharstr's to the value given as 'length.' It does

not advance through the data structure. The type and value results are unspecified, but 'ptr' is correct.

type = rlistnop(102)

This does nothing. The type for the next element is returned, but the value is unspecified, and 'ptr' is correct. It can be called to obtain the current position without advancing and/or get the type of the next element without actually reading it.

If 'dest' is non-zero AND 'type' is charstr, then integer, index, empty, or boolean 'length' elements are read and stored at location 'dest'. In that case an ABORT is generated if the next 'length' elements are not of type 'type'.

If 'dest' is non-zero, 'type' may not be pcplist type. That is, only non-list elements may be stored in a designated array.

The result type and value are summarized here:

rtype: value

----: ----

list: number of elements

index: the index value

integer: the address of 32 bits

(On 11, first word is most significant)

charstr: the address of the string

boolean: TRUE or FALSE

empty: zero

bitstr: address of bitstring

Notes:

Currently, the ABORT takes the form of a procedure call to err($"Bad PCP data type -RLIST"); .

Examples:

```
OPENPORT rlist(&params, zone:  [iport])
% read one element of type index. put it in indexvalue %
  PCALL [iport](pcpindex, 0:  indexvalue)
% read three booleans into array ary %
  PCALL [iport] (pcpboolean, $ary, 3);
% read one element, either index or list %
  type _ PCALL [iport] (0:  value)
    CASE type of
      =pcpindex:  ... %value is in 'value' %
      =pcplist:  ... % length is in 'value' %
        % read entire list of charstrings into 'str' array
        %
          PCALL [iport] (pcpcharstr, $str, value);
      ENDCASE err($"wrong type element");
```

Writing a PCP Data Structure

```
WLIST (adr REF, n -> [oport])
```

Openport on 'wlist' takes the address of a block in which to build the data structure and a word count representing the number of words available in the block. A HELP signal requests more room if the block is overrun.

Each subsequent PCALL on oport builds one element in the data structure (approximately). The PCALL arguments are type and value. WLIST always returns a WORD count and a pointer.

```
count _ PCALL [oport] (type, value:  ptr);
```

'type' is a PCP data type or other special operator.

'value' is usually the value of the PCP element.

´count´ is a WORD count of the structure so far.

´ptr´ is a pointer that can be used to reset the position in the data structure.

The types allowed and the action taken are as follows.

type: action

----: ----

pcpindex:

Element of type index, value ´value´ is constructed.

pcpboolean:

Element of type boolean is constructed.

(value=Ø = FALSE)

pcpempty:

Empty element is constructed.

pcpinteger:

´value´ points to 32 bits used to make integer element. On 11, most significant 16 bits is first word.

pcpcharstr:

´value´ is the address of an L1Ø string. An element of type charstr (containing that string) is constructed.

pcpbitstr:

´value´ is the address of a bitstring. The first word of the bitstring is taken as an integer, which is the number of bits in the bitstring.

pcplist:

If ´value´ is zero, a list of unknown length may be built. Otherwise the list length is taken as ´value´ and appropriate checks are made.

pcppad:

One PCP-PAD element is constructed.

wlistend(200):

This closes the last list construction. If the length was provided, a length check is made and err is called if not correct. Otherwise the length is computed and stored in the list element.

wlistreset(201):

This resets the writing position to 'value', which must have been obtained from wlist previously as a 'ptr'. The next element written will follow the last element written when the 'ptr' was obtained.

This is dangerous! After doing this, you may NOT close a list (wlistend) that was started BEFORE the wlistreset was done. The 'count' after doing a wlistreset will be the number of words to the current position in the list, not the total number of words in the data structure.

wlistnop(202):

This is a NO-OP that returns 'count' and 'ptr' for the current position.

Note that lists may be nested and data structures may be built without prior knowledge of the contents.

If WLIST overruns the area:

A HELP(wlistoverflow, address, needed) is generated,

where 'address' is the address of the first word of the area, and 'needed' is the number of words that MUST be present to write the next element.

The proper return is RESUME(gothelp, newaddress, n),

where 'newaddress' is the address of the relocated area, and 'n' is the number of words allocated in that new area.

The helping routine must copy the entire area into the new area.

USER PROFILE DATA STRUCTURE AND TOOL

Introduction

User Profile refers to the per node data base that holds parameters describing the desired tool-independent characteristics of the FE. The dynamic data base is read when a node session begins and is used throughout the session to give the user a personalized FE.

The User Profile Tool is a separate, fully split NSW tool that is accessed through the Runtool command given to the WM EXEC. This tool gives the node the ability to modify his own User Profile data base.

Current Capabilities

The User Profile data base is not currently supported by either the WM or the FE. If the FE did read the User Profile data base, any modifications made in the User Profile Tool would become active at the next session, when the User Profile would again be read by the FE. Currently the User Profile Tool reads and writes the data base to a Tenex file that is uniquely named (using the project-node name).

Envisioned Capabilities

While the User Profile has never been fully integrated into the NSW, it is envisioned that the User Profile be a data base whose access is restricted by the WM (e.g. as an NSW file or through WM calls to read/write it). At session startup, it would be read by the WM and returned as a PCPB8 data structure as a result of WMLOGIN.

The User Profile Tool would provide immediate profile updating and the option to make the modification permanent or restrict it to the current session. To support immediate profile updating, the FE would provide an external call, allowing it to read a profile or profile-part from the User Profile tool.

User Profile Data Structure

Data Structure

The User Profile data structure consists of one list (coded in PCPB) containing the following five elements (in order):

1. Profile - BITSTR

   Feedback, herald, etc. See definition of fields below.

2. Startupstring - CHARSTRING

   Startup input command string.

3. Tool list - LIST(tool1, tool2, ...)

   Each element tool-i is of type CHARSTRING and contains a legal name of a tool.

4. Control Character list - LIST (Referred to as cntchr list.)

   Each element of the list is itself a LIST (referred to as dvclst) of the following structure:

   LIST(index, LIST(cf,char,echo), LIST(cf,char,echo), ...)

   where:

   index - INDEX - device code

   cf - INDEX - Control function code

   char - CHARSTRING - String of characters where each serves the specified function

   echo - CHARSTRING - String to echo when the control character is typed.

5. Version - INDEX

   For compatibility check.

Data Structure Conventions

1. The profile bitstr is currently 23 bits long, with the bits allocated as follows (bit number 1 is the leftmost):

Field	Bits
feedback length	1 thru 8
herald length	9 thru 12
recognition mode	13 thru 14
secondary recognition	15 thru 16
prompting	17 thru 18
command word length	19 thru 23

In the L10 programming language this results in the following record:

```
(prfl) RECORD

  padding[9], cmdwdlen[5], prpt[2], rcg2[2], rcg[2],
  hldlen[4], fblen[8] ;
```

2. Startupstring is a null string if none is specified.

3. Tool-list convention:

First element is the entry tool, or NULL if none is defined. (Undefined entry tool causes the user to stay in the EXEC after login.)

Other elements (if any) are tool names (all as strings).

4. Control characters:

If a list for a specific device (dvclst) does not exist all control characters default.

If a list exists for a specific device only those control-characters that deviate from the default have entries.

If this entire control character list has only one NULL element all control characters for all devices default.

5. Version number is currently 9.

Meaning of Fields in the Profile Bitstr

Prompt

Ø = Verbose (default value)

1 = Terse

2 = Off

Recognition (Both Levels)

Ø = Anticipatory

1 = Terse (default value)

2 = Fixed

3 = Demand

Feedback length, herald length, and command word length contain the corresponding length (in characters) and default to the maximum number allowed.

Control Function Indexing and Defaults

Function	Index	Default
COMMAND ACCEPT	4	^D
COMMAND DELETE	24	^X
REPEAT	2	^B
BACKSPACE CHARACTER	8	^H
BACKSPACE WORD	23	^W
BACKSPACE STATEMENT	16	^P
LITERAL ESCAPE	22	^V
IGNORE	Ø	No Default
SHIFT CHARACTER	47	No Default
SHIFT WORD	92	No Default
TAB	9	^I
OPTION	21	^U

Device Code Indexing

Device Name	Index
TI	2
NVT	3
LINEPROCESSOR	4
IMLAC	5
EXECUPORT	6
TTY33	7
TTY35	8
TTY37	9

GENERATING A NEW FRONTEND

Introduction

The following steps must be taken to create a new Frontend:

- Make sure the relocatable binary files are up to date.

- Load the desired Frontend configuration to create a save (.SAV) file.

- Create the initial grammar.

- (Other steps may be necessary depending on the Frontend configuration.)

The actual loading process is usually done by a RUNFIL program, with a RUNFIL file available for each Frontend configuration. All compiling, loading, and saving operations currently must be done on a TENEX or TOPS-20 host.

Compilation

Each Frontend source file contains the name of the compiler(s) and the REL file(s) that are to be used when compiling that source file. To create a Frontend for the PDP-10, use the L10 compiler; for the PDP-11, use the L10ll compiler. Of course, if the source file has not been changed since the existing REL file was created, it is not necessary to compile that source file before creating a new Frontend.

Frontend source files are currently NLS files. To compile them, use the Complie File command in the Programs subsystem. Sequential files may be compiled by simply running the same compiler as a TENEX subsystem, giving the sequential file as input, and specifying the REL file as output.

Loading

To load a Frontend, run a loader to bind all the REL files together and save the core image in a SAV file. For a PDP-11 Frontend, you must also format the SAV file into PDP-11 load format. The names of the RUNFIL files that perform the loading for each Frontend configuration are specified below, along with the names of the various files that comprise each Frontend module.

To make a Frontend ready for use, the SAV file is placed in the file directory in which it is to run. The initial grammar is placed in the same directory with the name EXEC.CGR; parse function files are also placed in the directory, with the extensions .PFC (for code) or .PFD (for data). This is explained in the "Grammar Compilation and Compaction" section of A GUIDE TO THE CML AND CLI.

In some cases, the Frontend is then ready to use. The exceptions are a stand-alone single fork tool that uses the Frontend and a Frontend that uses the shared page communication medium. In the first case, refer to "Making a Stand Alone Tool" in Appendix 2 of A GUIDE TO THE CML AND CLI. For the second case, the SAV file for the tool backend must be placed in the same directory as the Frontend, along with the Frontend and the initial grammar.

When making a PDP-11 Frontend, an additional step is performed by the RUNFIL file. The SAV file is converted into PDP-11 loading format, by way of program SAVBIN. The resulting BIN files are then loaded on the PDP-11. Because that loading process is still undergoing changes it will not be described in detail at this time.

Frontend Files

Below is a list of the Frontend source files, the REL files they produce, and information about which Frontend configuration requires them. When no file directory is given, the directory is <NSW-SOURCES>. Where several REL files are produced from one source, they are separated by a semi-colon (;). The L10 runtime support files (code and data) are not included in this list, but they are loaded in each Frontend configuration.

NEWCLI.NLS

NEWCLI.REL; required for ALL PDP-10 FEs

<L10|1>CLI.REL; required for ALL PDP-11 FEs

L10|1STGMGT.NLS

FESTGMGT.REL; required for ALL PDP-10 FEs

<L10|1>FESTGMGT.REL; required for ALL PDP-11 FEs

XOSI-CLI.NLS

XOSICLI.REL; XOSIDATA.REL; required for ALL PDP-10 FEs

CGPFADDS.NLS

PFADDS.REL; required for ALL PDP-10 FEs

<L10l1>PFADDS.REL; required for all PDP-11 FEs

XFEROUTINES.NLS

XFERTNS.REL; required for ALL PDP-10 FEs

XFEDATA.NLS

XFEDATA.REL; required for ALL PDP-10 FEs

<L10l1>FEDATA.REL; required for all PDP-11 FEs

DPYPKG.NLS

DPYPKG.REL; required for ALL PDP-10 FEs

<L10l1>DPYPKG.REL; required for ALL PDP-11 FEs

DPY-10.NLS

DPY-10.REL; required for all PDP-10 FEs

MSG-3I.NLS

MSG-3I.REL; MSG-3DATA.REL; required for MSG-3

TYPECLI.NLS

TYPEI.REL; required for TYPEOUT

SAFE.NLS

SAFEI.REL; required for Stand Alone FE

<RELNINE>NLSI.NLS

<RELNINE>NLSI.REL; required for Shared Page

PCPB8-10.NLS

PCPB8.REL; required for MSG-3, Raw Net Conn. and Shared Page

<L1Ø11>PCPB8-11.NLS

<L1Ø11>PCPB8.REL; required for all PDP-11 FEs

CGRAMLDR.NLS

CGRAMLDR.REL; required for ALL PDP-1Ø FEs

<L1Ø11>DPY-11.NLS

<L1Ø11>DPY-11.REL; required for all PDP-11 FEs

DPYDATA-1Ø.NLS

DPYDATA.REL; required for all PDP-1Ø FEs

<L1Ø11>DPYDATA-11.NLS

<L1Ø11>DPYDATA.REL; required for all PDP-11 FEs

MSG3FE.RUN

RUNFIL input to make MSG-3 Frontend

TYPECLI.RUN

RUNFIL input to make TYPEOUT Frontend

<RELNINE>NLS9FE.RUN

RUNFIL input to make Shared Page Frontend

SAFE.RUN

RUNFIL input to make Stand Alone Frontend

<L1Ø11>CLI.RUN

RUNFIL input to make PDP-11 Frontend

REFERENCES

1. Donald I. Andrews, Beverly R. Boli, and Andrew A. Poggio, A Guide to the Command Meta Language and Command Language Interpreter. Augmentation Research Center, Stanford Research Institute, Menlo Park, California. February 3, 1977. (28744,).

2. Donald I. Andrews, 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,).