CTAM™ APPLICATION PROGRAMMER'S GUIDE

Copyright © 1988 by Convergent, Inc., San Jose, CA. Printed in USA.

First Edition (Nov 1988) 73-00515-A

All rights reserved. No part of this document may be reproduced, transmitted, stored in a retrieval system, or translated into any language without the prior written consent of Convergent Technologies, Inc.

Convergent makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Convergent reserves the right to revise this publication and to make changes from time to time in its content without being obligated to notify any person of such revision or changes.

Convergent, Convergent Technologies and NGEN are registered trademarks of Convergent, Inc.

Art Designer, AutoBoot, AWS, Chart Designer, ClusterCard, ClusterNet, ClusterShare, Context Manager, Context Manager/VM, CTAM, CT-DBMS, CT-MAIL, CT-Net, CTIX, CTOS, CTOS/VM, CWS, DPL, Document Designer, GT, IMAGE Designer, IWS, MiniFrame, Network PC, PC Emulator, PC Exchange, Phone Memo Manager, PT, S/50, S/120, S/320, S/640, S/1280, S/Series, Series/286i, Series/386i, Server PC, Shared Resource Processor, Solution Designer, SRP, TeleCluster, The Cluster, The Operator, Voice/Data Services, Voice Processor, WGS/Calendar, WGS/DESKTOP, WGS/Mail, WGS/Office, WGS/SpreadSheet, WGS/WordProcessor, WorkGroup Servers, and X-Bus are trademarks of Convergent, Inc.

UNIX and RFS are trademarks of AT&T.

This manual was prepared on a Convergent Technologies S/320 Computer System and printed on an Apple LaserWriter II Printer.

Contents

1 **Overview** **1-1**
What Is CTAM? 1-1
Who Should Use This Manual? 1-3
How This Manual Is Organized 1-3
Conventions Used in This Manual 1-4
Related Documentation 1-5

2 **CTAM Application User Interface** **2-1**
What Are Forms and Menus? 2-2
Moving Within and Between Forms and Menus 2-3
Performing Other Functions 2-4

3 **Using the CTAM Windowing System** **3-1**
Overview of the Window Terminal Concept 3-1
CTAM Window Manager Files 3-4
Language Dependent Files 3-6
Terminal Description Files 3-6
CTAM Terminal Support 3-12
Adding a New Terminal 3-13

4 **Introduction to DPL Programming** **4-1**
The DPL Form and Menuing Tools 4-2
Using the Dialogue Interpeter 4-2
Resource File Example #1: 4-2
Resource File Example #2: 4-4
DPL Entities: Forms, Fields, and Items 4-5
Forms 4-7
Fields 4-8
Menu Fields 4-10

List Fields 4-12
Edit Fields 4-13
Text Fields 4-17
Items 4-17
Item Attributes 4-18
Item Values 4-19
Events 4-20
Scoping of Events 4-22
The Onkey Event 4-23
Action Routines 4-24
Event Language Control Flow 4-29
Variables 4-33
Defining Global Variables 4-37
Special Variables 4-38
Summary of Terms 4-40

5 Programmatic Forms and Menus 5-1
Compiling a DPL Program 5-2
The Base Set of Forms Calls 5-2
Example 1 5-2
Example 2 5-3
Using the Forms Compiler 5-5
Special Features 5-5
Restrictions 5-6

6 The CTAM Internationalization Kit 6-1
Internationalization Subroutines 6-1
Nationalization Files 6-9
Language Configuration Database 6-10

7 Using CTAM with COBOL, BASIC, and FORTRAN 7-1
COBOL 7-1
BASIC 7-3
FORTRAN 7-4

APPENDIXES

A Introduction to the CTAM Manual Pages **A-1**

Glossary **Glossary-1**

Index **Index-1**

LIST OF FIGURES

Figure		*Page*
1-1	The CTAM Window Manager	1-2
2-1	Sample Form #1	2-2
2-2	Sample Form #2	2-3
3-1	The User/Kernel Interface Under CTAM	3-2
3-2	Using **terminfo** Description Files	3-7
4-1	Sample Form #3	4-3
4-2	Sample Form #4	4-4
4-3	Sample Form #5	4-5
4-4	Hierarchy of DPL Entities Under CTWM	4-6

LIST OF TABLES

Table		*Page*
3-1	Default CTAM Keyboard Mapping	3-9
3-2	Released Terminal Support	3-13

1

Overview

What Is CTAM?

The Convergent Terminal Access Method (CTAM) package consists of a number of tools to aid software developers in the rapid creation and integration of easy–to–use applications that use multiple, concurrently active windows on low–cost ASCII terminals. Using CTAM, developers can create new applications with intuitive, consistent formats, or integrate existing programs into a windowing environment.

Among the components in the CTAM package, the Dialogue Programming Language (DPL) is a simple description language used to describe *forms* and *menus* that are displayed in windows. (Chapter 2 discusses forms and menus in detail.) Specifically, DPL describes two things:

- How a form is to appear on the screen: where on the screen, how large a window, what text is displayed in menu selections, which attributes (such as highlighting) are turned on.
- What actions are to be initiated when specific selections are activated: display another form or menu, run a shell script or other program.

Tools in the CTAM package include:

ctwm(1W) This program provides interactive multi–process window management, enabling applications to run concurrently in movable, resizable windows on ordinary ASCII terminals with no programming required.

dplrun(1) This runtime program provides forms interpretation and runs the stand–alone forms and menus specified in DPL. Use **dplrun** to quickly create a menu–driven interface for multiple applications and utilities.

rcc(1) This DPL compiler can be used to compile forms and menus into an unalterable form for security or other reasons.

Libraries (under **/usr/lib**) in the CTAM package include:

libctam.a This library of routines is used to create, display to, and manipulate concurrent windows.

libdpl.a This library is used to add DPL–style forms and menus to an application, programmatically.

libxnls.a This library of routines is used to develop applications that work using a variety of natural languages.

The DPL forms and menuing system, along with the ASCII terminal windowing system, comprise the CTAM applications development kit, designed to provide a complete platform for end user presentation software on a CTIX system.

The CTAM windowing system is designed to be extensible to work with terminals not directly supported in the released product. Support for new terminals can be added by creating or modifying the description files discussed in Chapter 3.

Figure 1–1 illustrates the role the CTAM window manager (**ctwm**) plays in converting ANSI x3.64 generic data received from an application into terminal specific output. ANSI x3.64 is a standard set of character sequences and escape codes for computer terminals. Data received from the keyboard is passed to the application untouched.

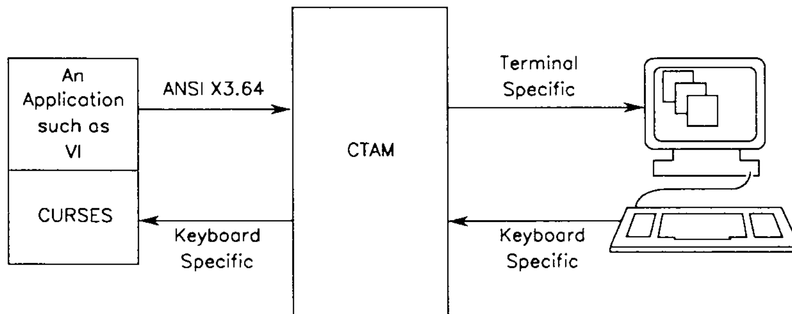

Figure 1-1. The CTAM Window Manager

Who Should Use This Manual?

The primary audience for this manual consists of programmers and system integrators familiar with UNIX and C who want to develop form– and menu– driven applications and take advantage of the multiple windowing aspects of CTAM.

This manual is also intended for readers adding support for a new terminal or for anyone interested in customizing the operation of CTAM–based products.

How This Manual Is Organized

This chapter gives a brief overview of CTAM, an outline of the manual's contents, and a description of conventions used throughout the manual.

Chapter 2, "CTAM Application User Interface," explains the basics of selecting and activating forms and menus in CTAM–based programs.

Chapter 3, "Using the CTAM Windowing System," describes CTAM's components in detail and provides an example procedure for adding a new terminal.

Chapter 4, "Introduction to DPL Programming," explains how to create and manipulate forms and menus using the Dialogue Programming Language.

Chapter 5, "Programmatic Forms and Menus," discusses ways in which programs written in C or other languages can interact with forms and menus.

Chapter 6, "CTAM Internationalization Kit," describes how to develop software that works using a variety of natural languages.

Chapter 7, "Using CTAM with COBOL, BASIC and FORTRAN," outlines what you need to do to use the CTAM libraries with programming languages other than C.

Appendix A contains CTAM–related manpages that are not included in the *CTIX Operating System Manual.*

Conventions Used in This Manual

The following conventions are used throughout this manual:

- Italics (*default*) indicate either a word that is displayed on the screen, or a word that is described in the Glossary.
- Boldface indicates literal characters, like those in the command **dplrun**, which appear or must be typed exactly as shown.
- Boldface characters followed by a number in parentheses, such as **dplrun**(1) or **sh**(1), are CTIX commands or file formats that are described in Appendix A, or in either the *CTIX Operating System Manual* (S/MT systems) or the *CTIX/386 Operating System Manual* (Server PC systems).
- Boldface with the first letter capitalized (**Return**) refers to the terminal key to press.

  Since DPL applications displaying forms and menus run on many terminal types (and therefore, many different keyboards), this manual makes use of *virtual* key names to describe terminal keys. For example, instead of the virtual key **Enter**, you might press **Go, Do**, or **Linefeed**, depending on your terminal type. A complete list of virtual keys is given in Table 3–1, "Default CTAM Keyboard Mapping."
- A preceding caret (^) in a keystroke sequence indicates a *control code*; hold down the **Control** key (or **Code** key on a PT/GT terminal) and press another key simultaneously. For example, **^Tab** means press the **Control** key, and while holding it down, press the **Tab** key.
- The dash (–) indicates that you should press two keys at the same time. For example, **Shift–Help** means press both the **Shift** and the **Help** key simultaneously.

Related Documentation

For further information about CTIX programming, consult the *CTIX Programmer's Guide* and the *Programmer's Guide: CTIX Supplement.*

2

CTAM Application User Interface

One of the tools in the CTAM software development package is the Dialogue Programming Language (DPL), a high level language for developing *form*– and *menu*–based applications. Forms and menus are displays on the screen containing items that can be selected by pressing a specified keystroke sequence. A *dialogue* is a session in which an end user interacts with forms and menus to get services provided by one or more application programs.

Chapter 4 discusses how to write programs in DPL. This chapter describes how to interact as an end user with applications written in DPL; the following tasks are described:

- How to move the cursor within and between forms and menus
- How to select items from menus
- How to fill in edit fields
- How to invoke help text, if available
- How to handle minor error situations

Since DPL applications displaying forms and menus run on many terminal types (and therefore, many different keyboards), this manual may not describe the keys that exactly match your terminal. Where appropriate, a set of key names from various terminals are described; the primary key names given are virtual (generic) key names. For a complete list of virtual keys, see Table 3–1, "Default CTAM Keyboard Mapping."

What Are Forms and Menus?

A *form* is a display containing one or more *fields*. Fields vary in shape and size, depending on the number and format of the items inside. Some fields take up the whole screen; others take up enough space for only a few characters. Depending on the type of field, items in a field can be selected or edited, or if the field is for viewing only, neither selected nor edited.

An *item* is a piece of text within a field. Normally, items appear one per line in a single column.

In comparison to a form, which can contain more than one type of field, a *menu* is a single field containing selectable, noneditable items. In other words, you can choose an item on a menu list, but you cannot change any of the items on the list. A menu can be part of a form, but a form is never part of a menu.

An example of a form containing various types of fields is displayed in Figure 2-1. Note that when there are more items in a field than will fit within the coordinates specified for the field, the system displays a *scroll bar* to the right of the menu or text field to indicate that there are more items than those shown. The scroll bar is a vertical highlighted bar with arrows indicating your relative position on the list of items.

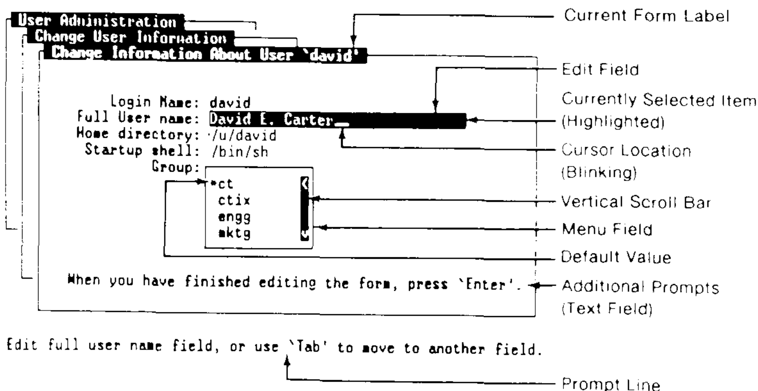

Figure 2-1. Sample Form #1

Moving Within and Between Forms and Menus

In the example form below, the user is asked to make selections in three fields: to select an entree, to fill in a wine choice, and to select a dessert. The entree and dessert fields are examples of *menu fields*, which contain selectable but noneditable items. The wine field is an example of an *edit field*, which can be both selected and edited. The prompt at the bottom is an example of a *text field*, which cannot be selected or edited.

```
Today's Lunch Menu

    Select an Entree:         *Beef
                               Chicken
                               Fish

    Enter a Wine Selection: [                    ]

    Select a Dessert:         *Pie
                               Cake
                               Fruit

    Make your selections and press 'Go' to execute,
    or press 'Cancel' to exit without executing the form.
```

Figure 2-2. Sample Form #2

The *arrow keys* (**Up**, **Down**, **Forward**, and **Back**) and the **Tab** and **Return** keys move the *cursor* and make selections on the screen. In the above example, you would move the cursor within the entree menu field (using **Up** and **Down**) to make a selection. To save your entree selection and move the cursor to the wine field, you would press the **Tab** key.

You would then type a wine name and press **Return**. As with the **Tab** key, **Return** moves the cursor to the next field. You would then make your dessert selection (using **Up** and **Down**).

In edit fields, the **Backspace** key deletes the character to the left of the cursor. The **Forward** and the **Back** keys move the cursor without deleting existing text. The **Delete** key deletes the character at the current cursor position. To insert text in the middle of what you have already entered, move the cursor to where you want to insert and enter the text.

Fields with more than one item often contain preselected or *default* values that can be changed by moving the cursor to another item in the field. Currently selected items are represented either by a highlighted bar or by an asterisk (*) that the system places to the left of selected items. To see items that are off the screen in a scrollable field (as in Figure 2–1), use **Down** to force the list to scroll. Remember that to move to another field without changing your selection in a menu field, you move the cursor using **Tab** or **Return**.

NOTE

> For more advanced users, a shortcut is available for selecting items in a menu field. Instead of striking the arrow keys a number of times, you can enter the first few letters of the choice that you desire, and the cursor moves to that choice. For the above example, if the cursor is located in the entree field, entering the letter 'F' moves the cursor to *Fish*.

Multiple selection menu fields, also called *list fields*, allow you to make more than one selection. (There are no list fields in the above examples.) When a menu field allows multiple selections, the screen provides instructions on how to make your choices. Usually you press the **Mark** key (labelled **Select** on many other keyboards) to indicate your choices. When you press **Mark**, usually an asterisk appears next to your choice. To 'unmark' something if you change your mind on a selection, press **Mark** on that item a second time.

Performing Other Functions

To indicate that you are satisfied with your selection and that you want it to be processed by the system, press the **Enter** key, also called **Go**, **Do**, or **Linefeed** on supported terminals. Some applications ask you to press the **Finish** key (also entered as **^D** on some terminals) when you have completed filling out a form. (Recall that **^D** means that you hold the **Control** key down while you strike the letter 'd'.)

The **Cancel** key (also entered as **^X** on some terminals) allows you to exit a form or menu *without executing* the currently selected items.

In many form– and menu–based applications you can press **Help** (or the appropriate key for your terminal) to display more information about a function.

If characters on the screen are ever displayed improperly, such as broken lines around a menu, or unreadable characters on the screen, try typing **^L**; this control code instructs the system to repaint the screen. If the screen is still unreadable, report the problem to your system administrator.

If the system ever ignores your input, for advanced users, the ^\ sequence instructs the system that you would like to exit the application you are running. Note that this operation may cause the task being performed to be left in an incomplete state, and should be used only as a last resort.

3

Using the CTAM Windowing System

This chapter describes the various components of the CTAM windowing system and their places in the CTIX file system. Read this chapter if you are interested in either adding support for a new terminal or learning more about CTAM internals.

Overview of the Window Terminal Concept

The CTAM Window Manager [**ctwm**(1W)] is a program that enables multiple applications to run concurrently in movable, resizable windows on an ordinary ASCII terminal. A *window terminal* is a software construct presented to an application process by the window manager.

The window terminals used by applications running under the window manager are provided by the *window driver*, **wxt**. The window driver handles all communication between the application processes and the physical terminal. Window devices (**/dev/wxt/wnnn**) are used to communicate with the terminal instead of tty devices (**/dev/ttynnn**).

Figure 3–1 illustrates the mechanics of CTAM at the user/kernel interface for applications running under the window manager.

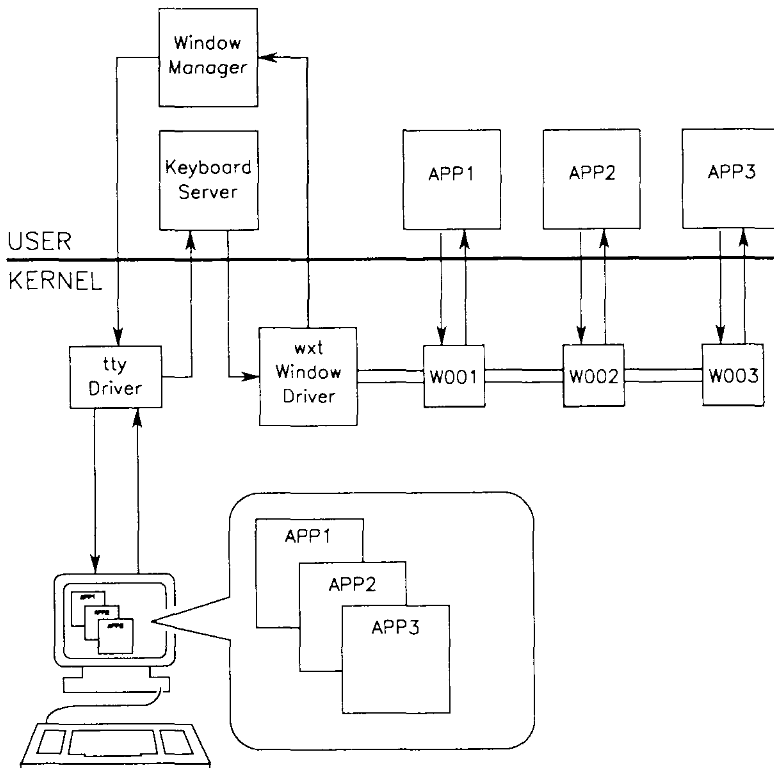

Figure 3-1. The User/Kernel Interface Under CTAM

In Figure 3–1, the keyboard server loops, reading from standard input and packaging keystrokes; then it transmits the keystrokes to the window driver by means of an **ioctl**(2). The window driver acts as a multiplexer, routing data for window traffic. Keystroke input is unpackaged and queued on appropriate CLISTs to be read by the applications via standard system *read* calls.

In certain modes [see the discussion of **keypad** under **wgetc**(3W)], virtual keys described in the **kbmaps**(4W) file for the terminal are captured and replaced with the *internal value* of the virtual key as defined in **/usr/include/kcodes.h**. In other modes, all keyboard input is passed unchanged to the user process. (For more information on

kbmaps and **kcodes.h**, see "Terminal Description Files," later in this chapter.)

Output from applications is packaged by **wxt** along with a window identifier and is routed to the window manager by means of an **ioctl**.

The window manager controls the screen, interpreting escape sequences, translating screen controls (such as attribute selection and cursor positioning), and determining how to layer output from **wxt** on the physical terminal based on what part of each window is visible. The window manager can also be used in interactive mode to create new windows, size and move existing windows on the screen, perform cut and paste, and so on.

Applications linked with the CTAM library (**libctam.a**) contain the code necessary to display windows on the screen without using the window manager. During initialization, the library finds out if **ctwm** is running. If not, the library itself interprets outgoing escape sequences and layers windows on the final output, writing the window border characters and the visible contents of each window. If **ctwm** is running, the library sends output and special information requests to **wxt** using an **ioctl**.

During initialization, the window manager modifies the user's **$TERM** variable, appending the suffix **ctam**, so that, for example, the **$TERM** variable **to300** is changed to **to300ctam**. As a result, applications running in a window that make use of **curses**(3) send the window manager a generic (CTAM) set of escape sequences to be interpreted. On the output side, the window manager sends terminal–specific sequences to the screen, based on the original value of **$TERM**. The CTAM set of escape sequences is a subset of ANSI x3.64. [For more information on escape sequences, see **escape**(7W) in Appendix A.]

All processes using windows on a terminal can display at the "same" time. Several processes may be using different windows on one tty, but there can be only one controlling process, which receives keyboard data.

The window terminal structure means that not all escape sequences work in windows. Some terminal display activities are not supported in windows. For example, the vt220 double–height and double–width characters cannot be supported. [The supported sequences can be found in **escape**(7W).] The window manager filters out the control codes for unsupported commands and ignores them; other unsupported escape sequences may appear as an inverse video question mark.

Applications that attempt to directly open their controlling tty by device name do not work; however, applications can open **/dev/tty** or use **ttyname**(3C) and **isatty** to get the correct device name.

Programs that read and write on UNIX standard input and output function normally in a window environment. [For more information on **stdin** and **stdout**, see **stdio**(3S).]

As opposed to coding keystroke sequences directly into applications, all complex screen control should be done using **curses**(3); otherwise, applications run only on terminals that emulate ANSI x3.64. For example, in the line of code below, the escape sequence for highlighting a string of text is coded directly into the program:

```
printf("\E[2m Hello World \E[0m");
```

This works fine under CTAM; however, it is more efficient to perform the same task using **libctam.a**, because the code is applicable to a wider range of terminals. For example:

```
(curses initialization)
        .
        .
        .
wattron (win, A_REVERSE);
wprintf (win, "Hello World");
wattroff(win, A_REVERSE);
```

For more information on the files described in this chapter, see **terminfo**(4) in the *CTIX Operating System Manual*, and see **fonts**(4W) and **kbmaps**(4W) in Appendix A.

CTAM Window Manager Files

The CTAM window manager (**ctwm**) is a CTAM–based application program that works in conjunction with a loadable software device driver called **wxt**. In order for the window manager to function, **wxt** must have been loaded and there must be corresponding *special files* in /**dev**, listed below. Most systems should have **wxt** load automatically during boot time; the CTAM installation software ensures that this happens by modifying files under /**etc**, listed below.

/etc/lddrv/wxt.o

This file is the **wxt** driver object code. This driver may be loaded and unloaded manually with the **lddrv**(1) command. Normally, the **wxt** driver is configured to be loaded automatically at system boot time by the CTAM installation software.

/etc/master

The loadable driver must have an entry in **/etc/master** before it can be loaded. The CTAM installation process creates an entry in this file if none already exists.

/etc/drvload

During system startup, this shell script is executed to load and start any loadable device drivers used by the system. The CTAM installation process appends a line to this file to cause the **wxt** device driver to be loaded automatically each time the system is rebooted.

/dev/ttynnn

Each tty special file is named by "tty" followed by three decimal digits representing the device's minor number. These special files can be opened by programs. The operating system ensures that the corresponding **/dev/wxt/wnnn** file is accessed.

/dev/window

The special file **/dev/window** is used by the CTAM library to create new windows, when running under the CTAM window manager. When a process opens **/dev/window**, the **open**(2) is rerouted to an unused *window device*, described below. This special file has the *major number* of the **wxt** driver and the *minor number* zero.

/dev/wxt/wnnn

Each window special file is named by a "w" followed by three decimal digits representing the device's minor number. Only files named in this way work. The CTAM installation process automatically creates 255 of these files, the maximum number.

Language Dependent Files

There are two files that must be changed when nationalizing CTAM for use in a language other than English. These files are stored in a subdirectory of **/usr/lib/ctam** whose name is controlled by the **$LANG** environment variable. For example, if **$LANG** is defined as **french**, the files reside under **/usr/lib/ctam/french**. If the **$LANG** environment variable is not defined, CTAM applications default to the value **english_usa**. The directory **/usr/lib/ctam/english_usa** contains two files: **ctwm.rf**, which contains all of the prompts used by the CTAM window manager, and **dpl.rf**, which contains the messages and labels common to all DPL-based applications.

Terminal Description Files

The CTAM windowing system has been designed to be extensible to work with terminals not supported in the released product. (For a list of supported terminals, see Table 3–2.) To provide support for a new terminal, you must add a number of files to the system that describe various aspects of the terminal. The supporting files are described on the pages that follow.

When reading the names of the files listed, substitute the name of a particular terminal for the environment variable **$TERM**. Normally, applications that use either standard **terminfo** or standard **termcap** work in windows; applications using standard **termcap** use window terminal capabilities described in **/etc/CTWMtermcap**. Extended or customized versions of **terminfo** or **termcap** may not work without modification.

/usr/lib/terminfo/?/$TERM and **/usr/lib/terminfo/?/$TERMctam**

Two different **terminfo** files must be added for every terminal used with CTAM. First, the terminal must have a standard **terminfo** description file. Second, the terminal must have a CTWM **terminfo** description file with the name "ctam" appended. (The **?** denotes the first character in the terminal name.) For example, for a vt220 terminal, there should be a **/usr/lib/terminfo/v/vt220** file and a **/usr/lib/terminfo/v/vt220ctam** file. [For more information on **terminfo** files, see **terminfo**(4).]

There are two basic ways in which terminal description files are used; the first is by CTAM itself, and the second is by applications running within CTAM windows.

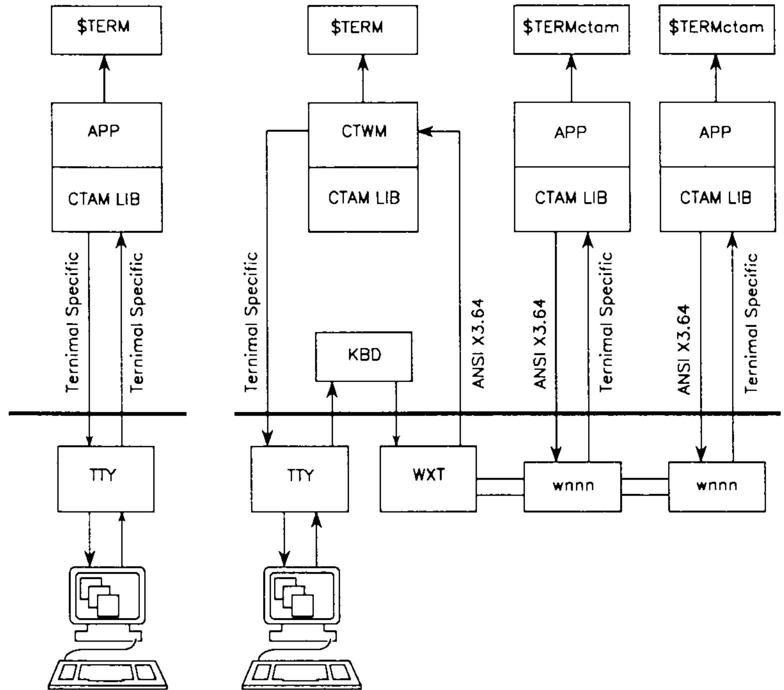

Figure 3-2. Using **terminfo** Description Files

As illustrated on the right in Figure 3–2, applications using the window manager receive *terminal specific* keystroke sequences from the **wxt** driver. Output from applications consists of ANSI x3.64 generic data taken from the terminal's CTWM **terminfo** file. When outputting to the screen, each application sends ANSI x3.64 data to the window manager, where the data is converted into terminal specific output using the standard **terminfo** file.

As shown on the left, applications that do not use the window manager receive terminal specific input and make use of the standard **terminfo** file to send terminal specific output.

The standard **terminfo** file describes the physical terminal, for the tty driver's use. The CTWM **terminfo** file describes the window terminal, enabling non–CTAM applications to work within windows. The CTWM **terminfo** file consists of keyboard definitions from the standard **terminfo** description file and output sequences from the CTAM description file, located in **$TERMctam.ti**, described later in this chapter.

/usr/lib/ctam/kbmaps/$TERM.kb

Each terminal used with CTAM should have a *keyboard description file*, which is an ASCII file consisting of three columns; white space between columns is ignored. The first column contains the names of CTAM virtual keys. A complete list of virtual key names is contained in the file **/usr/include/kcodes.h**. The second column specifies the byte sequence sent by the terminal when a specified key is pressed. Control characters can be specified using the same kinds of textual equivalents as those used in **terminfo** files with the **.ti** suffix. The third column is optional and contains the key name used in prompts displayed by some applications, such as the CTIX Administration Tools [see **adman**(1)]. The **kbmaps** files for the supported terminals listed in Table 3–2 provide examples of how to prepare keyboard description files. Below is an example listing of **/usr/lib/ctam/kbmaps/wy85.kb**:

```
#
# Keyboard map for Wyse 85
#

DleteChar      \177      shift-<X]
Backspace      \010      <X]
InputMode      \E[2~     Insert-Here
Enter          \E[29~    Do
Help           \E[28~    Help
Dlete          \E[3~     Remove
Mark           \E[4~     Select
Find           \E[1~     Find
Home           \E[27~    Home
F1             \E[17~    F6
F2             \E[18~    F7
F3             \E[19~    F8
F4             \E[20~    F9
F5             \E[21~    F10
F6             \E[31~    F17
F7             \E[32~    F18
F8             \E[33~    F19
F9             \E[34~    F20
```

Only those keys that have an obvious counterpart on the new terminal's keyboard need be defined. Virtual keys that are not defined have a default value. A listing of the default byte values (as well as keystroke equivalents and internal octal values) for all virtual keys is shown in Table 3–1. **METABIT** is a value assigned in **kcodes.h**.

TABLE 3-1

Default CTAM Keyboard Mapping

virtual key name	default sequence	character (keystrokes)	internal value (see kcodes.h)
Esc	\E	(Esc)	0033
Backspace	\010	(Backspace)	0010
BackTab	\E\t	(Esc Tab)	(0267 | METABIT)
Break	^]	(Ctrl-])	(0377 | METABIT)
Return	\r	(Return)	0015
Enter	\n	(LineFeed)	0012
F1	\E1	(Esc 1)	(0321 | METABIT)
F2	\E2	(Esc 2)	(0322 | METABIT)
F3	\E3	(Esc 3)	(0323 | METABIT)
F4	\E4	(Esc 4)	(0324 | METABIT)
F5	\E5	(Esc 5)	(0325 | METABIT)
F6	\E6	(Esc 6)	(0326 | METABIT)
F7	\E7	(Esc 7)	(0327 | METABIT)
F8	\E8	(Esc 8)	(0330 | METABIT)
F9	\E9	(Esc 9)	(0215 | METABIT)
F10	\E0	(Esc 0)	(0216 | METABIT)
s_F1	\E!	(Esc !)	(0241 | METABIT)
s_F2	\E@	(Esc @)	(0242 | METABIT)
s_F3	\E#	(Esc #)	(0243 | METABIT)
s_F4	\E$	(Esc $)	(0244 | METABIT)
s_F5	\E%	(Esc %)	(0245 | METABIT)
s_F6	\E\^	(Esc ^)	(0246 | METABIT)
s_F7	\E&	(Esc &)	(0247 | METABIT)
s_F8	\E*	(Esc *)	(0250 | METABIT)
s_F9	\E(	(Esc ()	(0376 | METABIT)
s_F10	\E)	(Esc))	(0376 | METABIT)
ClearLine	\Eci	(Esc c i)	(0331 | METABIT)
Creat	\Ecr	(Esc c r)	(0332 | METABIT)

<table>
<tr><th>virtual
key name</th><th>default
sequence</th><th>character
(keystrokes)</th><th>internal value
(see kcodes.h)</th></tr>
<tr><td>Undo</td><td>\Eud</td><td>(Esc u d)</td><td>(0333 | METABIT)</td></tr>
<tr><td>Find</td><td>\Efi</td><td>(Esc f i)</td><td>(0334 | METABIT)</td></tr>
<tr><td>Move</td><td>\Emv</td><td>(Esc m v)</td><td>(0335 | METABIT)</td></tr>
<tr><td>Dlete</td><td>\Edl</td><td>(Esc d l)</td><td>(0336 | METABIT)</td></tr>
<tr><td>Mark</td><td>\Emk</td><td>(Esc m k)</td><td>(0337 | METABIT)</td></tr>
<tr><td>Save</td><td>\Esa</td><td>(Esc s a)</td><td>(0341 | METABIT)</td></tr>
<tr><td>Redo</td><td>\Ero</td><td>(Esc r o)</td><td>(0342 | METABIT)</td></tr>
<tr><td>Rplac</td><td>\Erp</td><td>(Esc r p)</td><td>(0343 | METABIT)</td></tr>
<tr><td>Copy</td><td>\Ecp</td><td>(Esc c p)</td><td>(0344 | METABIT)</td></tr>
<tr><td>DleteChar</td><td>\177</td><td>(Del)</td><td>(0345 | METABIT)</td></tr>
<tr><td>InputMode</td><td>\Eim</td><td>(Esc i m)</td><td>(0346 | METABIT)</td></tr>
<tr><td>s_Move</td><td>\EMV</td><td>(Esc M V)</td><td>(0255 | METABIT)</td></tr>
<tr><td>s_DleteChar</td><td>\EDC</td><td>(Esc D C)</td><td>(0265 | METABIT)</td></tr>
<tr><td>s_Copy</td><td>\ECP</td><td>(Esc C P)</td><td>(0264 | METABIT)</td></tr>
<tr><td>Exit</td><td>^D</td><td>(Ctrl-D)</td><td>(0350 | METABIT)</td></tr>
<tr><td>Suspd</td><td>^W</td><td>(Ctrl-W)</td><td>(0351 | METABIT)</td></tr>
<tr><td>Cmd</td><td>^C</td><td>(Ctrl-C)</td><td>(0352 | METABIT)</td></tr>
<tr><td>Print</td><td>\Epr</td><td>(Esc p r)</td><td>(0353 | METABIT)</td></tr>
<tr><td>Beg</td><td>\Ebg</td><td>(Esc b g)</td><td>(0354 | METABIT)</td></tr>
<tr><td>Prev</td><td>\Epv</td><td>(Esc p v)</td><td>(0355 | METABIT)</td></tr>
<tr><td>Back</td><td>\Ebw</td><td>(Esc b w)</td><td>(0356 | METABIT)</td></tr>
<tr><td>Open</td><td>\Eop</td><td>(Esc o p)</td><td>(0361 | METABIT)</td></tr>
<tr><td>Rfrsh</td><td>^L</td><td>(Ctrl-L)</td><td>(0362 | METABIT)</td></tr>
<tr><td>Home</td><td>\Ehm</td><td>(Esc h m)</td><td>(0363 | METABIT)</td></tr>
<tr><td>Up</td><td>\Eup</td><td>(Esc u p)</td><td>(0364 | METABIT)</td></tr>
<tr><td>Down</td><td>\Edn</td><td>(Esc d n)</td><td>(0365 | METABIT)</td></tr>
<tr><td>Help</td><td>\E?</td><td>(Esc ?)</td><td>(0366 | METABIT)</td></tr>
<tr><td>Opts</td><td>\Eot</td><td>(Esc o t)</td><td>(0367 | METABIT)</td></tr>
<tr><td>Cancl</td><td>^X</td><td>(Ctrl-X)</td><td>(0370 | METABIT)</td></tr>
<tr><td>Page</td><td>\Epg</td><td>(Esc p g)</td><td>(0371 | METABIT)</td></tr>
<tr><td>End</td><td>\Een</td><td>(Esc e n)</td><td>(0372 | METABIT)</td></tr>
<tr><td>Next</td><td>\Enx</td><td>(Esc n x)</td><td>(0373 | METABIT)</td></tr>
<tr><td>Forward</td><td>\Efw</td><td>(Esc f w)</td><td>(0374 | METABIT)</td></tr>
<tr><td>s_Exit</td><td>\EEX</td><td>(Esc E X)</td><td>(0270 | METABIT)</td></tr>
<tr><td>s_Back</td><td>\EBW</td><td>(Esc B W)</td><td>(0276 | METABIT)</td></tr>
<tr><td>Close</td><td>\Ecl</td><td>(Esc c l)</td><td>(0301 | METABIT)</td></tr>
<tr><td>Clear</td><td>\Ece</td><td>(Esc c e)</td><td>(0302 | METABIT)</td></tr>
<tr><td>s_Home</td><td>\EHM</td><td>(Esc H M)</td><td>(0303 | METABIT)</td></tr>
<tr><td>RollUp</td><td>\Eru</td><td>(Esc r u)</td><td>(0304 | METABIT)</td></tr>
</table>

virtual key name	default sequence	character (keystrokes)	internal value (see kcodes.h)
RollDn	\Erd	(Esc r d)	(0305 | METABIT)
s_Help	\EHL	(Esc H L)	(0306 | METABIT)
s_Page	\EPG	(Esc P G)	(0311 | METABIT)
s_Forward	\EFW	(Esc F W)	(0314 | METABIT)

NOTE

Be sure that the key values for a new terminal do not collide with the values of other default or defined key values.

For example, if a terminal's **kbmaps** file defines **Down** as \ED, a collision condition exists with the default value of **s_DleteChar**. If the terminal must use the conflicting value, then its **kbmaps** file can redefine the value of **s_DleteChar** to clear the collision.

/usr/lib/ctam/fonts/$TERM.ft

Terminals that use multiple font sets (beyond standard ASCII) should have a *font description file* in **/usr/lib/ctam/fonts** to describe the alternate character set controls. The font description file is similar in format to a **.ti** file in **terminfo**. A keyword is followed by an equals sign "=", which is followed by a definition string and terminated by a comma or end-of-file.

For example, the file **gt.ft** is listed below:

```
smacs2=\E[12m,
rmacs2=^O,
ctline=1,
ctgraph=2,
smstrike=\E[9m,
decgraph=b(2d\0542fq2gd2yf2ze2|h2}M2~[2j\1361k`1l_1m]
          1x21qK1nL1tR1uS1vN1wM1,
```

For more information on font files, see **fonts**(4W) and the font files distributed with the CTAM release.

/etc/CTWMtermcap

If non–CTAM **termcap**–based applications are run in CTAM windows, the generic CTAM window terminal capabilities must be defined in **/etc/CTWMtermcap**. CTAM is released with a **/etc/CTWMtermcap** file describing supported terminals. Additional terminal descriptions can be added to **CTWMtermcap** in much the same manner as customized **$TERMctam.ti** files are created. (A sample listing of a **$TERMctam.ti** file is given at the end of this chapter.) When **ctwm** starts up, the **$TERMCAP** environment variable is set to **/etc/CTWMtermcap**.

/etc/termcap

All terminals used should have an entry in **/etc/termcap**, although CTAM applications do not actually use **/etc/termcap**. This is so that **tset**(1) correctly sets the **$TERM** variable for **termcap**–based applications when the user logs in.

CTAM Terminal Support

The eleven terminals supported as of CTAM Release 2.2 are displayed in Table 3–2:

TABLE 3-2

Released Terminal Support

Terminal	Description
ct235	Convergent Technologies TO–235
ct250, to250	Convergent Technologies TO–250
ct300, to300	Convergent Technologies TO–300
fortune	Fortune Systems 32:16 terminal
gt	Convergent Technologies Graphic Terminal
l220	Link 220
pt	Convergent Technologies Programmable Terminal
vt100	DEC VT100
vt220	DEC VT220
wy85	Wyse 85
bt970	Televideo BT–970 (RBOC custom terminal)

In most cases, software terminal emulators or emulation modes on other terminals work correctly when simulating one of the supported terminal types. Emulator packages often add features and functions not available on the original terminal. To take advantage of any additional features, the terminal support files must be customized as described in the section called "Adding a New Terminal," later in this chapter.

In particular, most VT100 emulation provides at least ten function keys. A real VT100 terminal has only four function keys. The keyboard values presented by the additional function keys vary in different emulation packages, so a modified **/usr/lib/ctam/kbmaps/vt100.kb** may be needed to correctly map the additional function keys.

Adding a New Terminal

In the example procedure below, a new terminal (a MicroTerm Act–IVA) is added to the system.

1. Terminals to be used with the CTAM window manager must have both a standard **terminfo** file and a CTWM **terminfo** file.

If there is no standard **terminfo** entry available for the terminal, create an entry by copying an existing entry and modifying it as described in **terminfo**(4). Sometimes a standard **terminfo** source file for the terminal is supplied by the terminal manufacturer. The terminal manufacturer's documentation is usually the best information source to use when writing the standard **terminfo** description.

If a standard **terminfo** (or **termcap**) entry exists for your terminal, but your terminal is not supported by CTAM, you can convert compiled **terminfo** (or **termcap**) entries into **terminfo** source. Refer to **captoinfo**(1M) and **infocmp**(1M).

NOTE

> As of Release 2.2, CTAM may not fully support terminals that reserve a blank character position for screen attributes. These terminals are called "non–hidden attribute" terminals.

CTAM requires a few specific **terminfo** capabilities for a usable terminal. The capability **clear_screen** (**clear**) must be defined. Cursor positioning is required, and can be provided by several different capability definitions. If **cursor_address** (**cup**) is defined, no other positioning capabilities are required. If **cup** is not defined, then the terminal must have:

cursor_up (**cuu1**) or **cursor_home** (**home**), and **cursor_left** (**cub1**) or **carriage_return** (**cr**), and **cursor_down** (**cud1**).

Terminals with **hard_copy** (**hc**) or **over_strike** (**os**) defined are not usable by CTAM. Other capabilities should be defined as appropriate to completely describe the terminal, but only the capabilities mentioned are required.

In this example, there already happens to be a description for the terminal called "act4". The compiled entry for this terminal resides in **/usr/lib/terminfo/a/act4**.

2. The CTWM **terminfo** file describes the window terminal and is used by applications to convert terminal specific input into generic ANSI x3.64 output.

   Prepare the CTWM **terminfo** file by merging your terminal's standard **terminfo** file keyboard descriptions with the window

terminal descriptions in the **$TERMctam.ti** file. Use **tic**(1) to compile the **terminfo** source into **/usr/lib/terminfo**. A listing of **$TERMctam.ti**, a skeleton CTWM **terminfo** source file, is given at the end of this chapter.

3. Check operation of the window manager. Once the **terminfo** entry is in place, the CTAM Window Manager should work. A simple test is to type:

   ```
   $ TERM=act4; export TERM
   $ ctwm
   ```

   The window manager should recognize the terminal and bring up a default window running a shell. Check to see how well everything works: Are the window borders solid or broken lines? Do any of the arrow keys or function keys work? If the screen becomes garbled, then the **terminfo** description needs work; if the borders or function keys do not work, but the screen is otherwise ok, then further steps must be taken.

   Other capabilities, such as line drawing, which enables **ctwm** to draw more sophisticated looking windows, should be defined to refine the output of the terminal. For more information on these options, refer to **terminfo**(4).

4. Create a keyboard description file for the terminal. Make a new file in **/usr/lib/ctam/kbmaps** for the terminal and name it **act4.kb** (the name of the terminal followed by ".kb"). At a minimum, put entries for the terminal's arrow keys and function keys in the file. Check the success of the new keyboard description file by rerunning the window manager and trying the function keys after typing **^Z**.

5. If the terminal uses multiple fonts, use a font file in **/usr/lib/ctam/fonts** to describe the alternate character set controls. In this example, the file should be called **/usr/lib/ctam/fonts/act4.ft**. Check the success of the new font description file by rerunning the window manager.

6. At this point, all CTAM applications (this includes applications such as the CTIX Administration Tools and WGS) should be fully functional. If non–CTAM applications are to be used under the window manager, then continue.

7. For **termcap**–based applications to properly recognize a terminal's arrow and function keys under the window manager, an entry must be made in **/etc/CTWMtermcap**. This is

considerably easier than adding the second **terminfo** description, since the **termcap** file is an editable ASCII file. Append to the file an entry called **act4ctam** that describes the terminal's arrow and function keys, followed by a "tc=ctam" entry that causes the rest of the entry to be the same as the "ctam" **termcap** entry.

The code for the example **terminfo** source is as follows:

```
# CTAM window terminal terminfo(4) source module.
# $TERM represents your terminal's name as normally set in your
# login profile. When running CTAM, the environment is modified
# as TERM=$TERMctam so applications use the window terminal
# capabilities rather than the physical terminal capabilities.
#
# ****************************************************************
# Modify terminal information lines below to match your terminal
# ****************************************************************
$TERMctam| <terminal under CTAM 2.1 window manager>
# keypad
list keypad capabilities from terminal's normal terminfo
description: kcub1, kcuf1, and so on.
# ****************************************************************
# The remainder of this description entry defines the display capabilities
# of the window terminal provided by the CTAM window driver, wxt, and
# should not be modified.
#
# booleans
msgr, am, xon,
# numbers
cols#80, lines#24,
# tabs
tbc=\E[3g, hts=\EH,
# navigation
cup=\E[%i%p1%d;%p2%dH, home=\E[H, ind=\ED, cr=^M,
cub1=^H, cud1=\E[B, cuf1=\E[C, cuu1=\E[A,
cub=\E[%p1%dD, cud=\E[%p1%dB, cuf=\E[%p1%dC, cuu=\E[%p1%dA,
nel=\EE, sc=\E7, rc=\E8,
# erasing, inserting, deleting
ech=\E%p1%dX,
clear=\E[H\E[2J, el=\E[K, ed=\E[J,
dch1=\E[P, dch=\E[%p1%dP, dl1=\E[M, dl=\E[%p1%dM,
ich1=\E[@, ich=\E[%p1%d@, il1=\E[L, il=\E[%p1%dL,
# bells, lights and whistles
bel=^G, sgr0=\E[m^O, smso=\E[7m, rmso=\E[m,
bold=\E[1m, dim=\E[2m, smul=\E[4, rev=\E[7m,
rmul=\E[24m, rmso=\E[27m, smso=\E[7m,
cnorm=\E[=C, civis=\E[=1C,
ldatt#6, smacs=^N, rmacs=^O, ldvl=x, ldht=q, ldul=l, ldur=k,
ldbl=m, ldbr=j,
# unsupported features
bw@, xsb@, xhp@, xenl@, eo@, gn@, hc@, hfl@, km@, hs@, in@,
da@, db@, mir@, os@, eslok@, xt@, hz@, ul@,
```

```
it@, lm@, xmc@, pb@, vt@, wsl@,
cbt@, csr@, tbc@, hpa@, cmdch@, mrcup@, ll@, cvvis@, dsl@,
hd@, blink@, smcup@, smdc@, smir@, prot@, invis@, rmcup@,
rmdc@, rmir@, flash@, ff@, fsl@, is1@, is2@, is3@, if@, ip@,
rmkx@, smkx@, smm@, rmm@,
pad@, indn@, rin@, pfkey@, ploc@, pfx@, mc0@, mc4@, mc5@,
rep@, rs1@, rs2@, rs3@, rf@, vpa@, ri@, sgr@, hts@, wind@,
ht@, tsl@, uc@, hu@, iprog@, mc5p@,
```

4

Introduction to DPL Programming

This chapter concentrates on the basics of programming in the Dialogue Programming Language (DPL), part of the CTAM applications development package; the following areas are covered:

- Different contexts in which DPL can be used
- Tools available for compiling DPL programs
- How to specify basic entities (forms, fields, and items)
- How to specify events
- How to specify action routines
- How to specify flow of control from one event to the next
- How to specify variables

Throughout this chapter, there are a number of programmatic examples to help you gain a working knowledge of the DPL system. The directory **/usr/lib/adman/english_usa**, used by the CTIX Administration Tools program, contains several more examples of DPL.

For a description of how to interact with DPL-based applications as an end user, see Chapter 2, "The CTAM Application User Interface."

The DPL Form and Menuing Tools

As described in Chapter 2, DPL is a high level language used to develop applications that display forms and menus. A *dialogue* is a session in which an end user uses keystroke sequences to interact with forms and menus.

DPL comes with form compilation tools [**rcc**(1)], a runtime interpreter [**dplrun**(1)], and programming libraries [**libdpl.a** and **libctam.a**] that allow DPL to be used in two kinds of contexts:

- As a description for a form that is used by an application program written in C, FORTRAN, Pascal, COBOL, or BASIC. In this context, which is described in Chapter 5, "Programmatic Forms and Menus," the application program makes calls to a forms library to interact with the user.
- As a description for a form that is accessed by a forms interpreter program (**dplrun**) called the Dialogue Interpreter. In this context, the form itself controls the session, commanding the Dialogue Interpreter to display additional forms or to run programs or shell scripts. Typically no programming other than DPL programming is done here.

Using the Dialogue Interpeter

For a session that uses the Dialogue Interpreter, the applications designer creates a forms definition file called a *resource file* (whose file suffix by convention is **.rf**). The resource file contains the form and menu definitions and the corresponding commands that drive the interpreter.

Resource File Example #1:

To see how easy DPL is to use, try the example below on your system. The DPL *keywords* used in the example are **form, onhelp, dohelp, field, menu, onselect, doexec, PopupForm,** and **text**. **Mainform** and **otherform** are names made up by the programmer, denoting names of forms. **$Enter, $Cancl** and **$Help** are *reserved variable names*, in this case describing keystrokes.

As in the C programming language, extra tabs and spaces are ignored in DPL code; thus, white space can be used to improve readability. All of the keywords, variable names, and other constructs used in the example are explained in the sections that follow.

```
form mainform "Professor Schmedlap's Bag of Tricks" (4,10)
     onhelp { dohelp( "/usr/lib/helpfile.hlp"); }
     field menu (2,2)
          "Run a Program"       onselect{ doexec( "ls" ); },
          "Display Another Form" onselect{ PopupForm( otherform ); },
          "Run a Shell Script"  onselect{ doexec( "sh","scriptname"); };
     field text (6,2)
          "Select the desired function and press '$Enter'.",
          "Press '$Help' for further information.";

form otherform "This Is Another Form" (6, 13)
     field text (2,2)
          "This is the text for the other form",
          "that is displayed when a menu item",
          "is selected.  This is a text field.",
          " ",
          "Press '$Cancl' to continue.";
```

The above example is executed from the shell like this:

```
$ dplrun file.rf
```

where **file.rf** contains the code listed above. The resulting display is a form as shown in Figure 4–1.

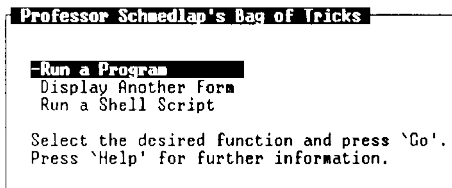

Figure 4-1. Sample Form #3

The label *Professor Schmedlap's Bag of Tricks* appears in the border at the top of the form. The upper left corner of the form is placed at coordinates (4,10) on the screen (row 4, column 10), where the origin (1,1) is at the top left corner of the screen.

The first menu item, *Run A Program*, is placed two rows, two columns from the upper left corner of the form, at (2,2), followed by the menu items *Display Another Form* and *Run A Shell Script*. These are the menu items that can be selected by the application user. At location (6,2) relative to the upper left corner of the form, there is a *text field*, a field that is displayed, but cannot be edited or selected.

If the first menu item is selected, the Dialogue Interpreter runs the program **ls**. If the second item is selected, an additional form is displayed. If the third item is selected, a shell script is executed. Figure 4–2 illustrates the resulting display if *Display Another Form* is selected.

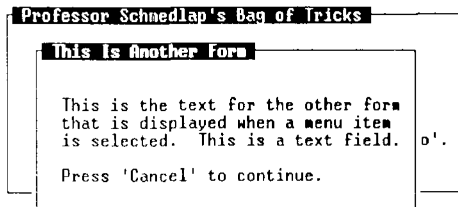

Figure 4-2. Sample Form #4

Resource File Example #2:

The previous example represents a menuing session, in which the user makes selections without editing any fields. Here is an example in which a user edits a field on a form:

```
form mainform "Personal Data" (4,5)
   onselect{ doexec( "echo", $mainform.MyField ); }

   field text (3,2) "Enter Your Name:";
   field MyField edit [boxed] (3,20)-(3,50) "";

   field text (5,2) "Select a Party:";
   field menu [chcksel] (5,20) "Republican", "Democrat",
                        "Anarchist", "Other ";

   field text (10,2) "Press $Enter to execute, $Cancl to exit.";
```

The resulting display, shown in Figure 4–3, is a form prompting the user to enter his name. The user input is then echoed on the screen by the **echo** command. The variable name **$mainform.MyField** is used much like a C structure field name, evoked by its use in the **doexec** command. The variable name consists of the form name, a dot, and the field name. The forms system fills in the value that the user enters or selects for that field.

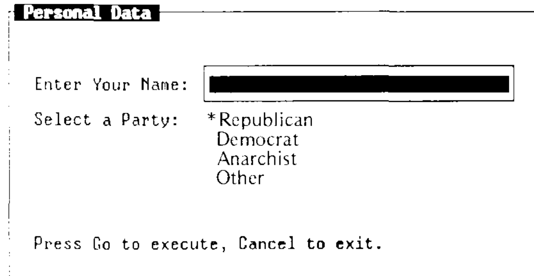

Figure 4-3. Sample Form #5

DPL Entities: Forms, Fields, and Items

There are three basic entities in DPL: *forms*, *fields*, and *items*.

The form is the basic unit of the DPL language. It is characterized by a box on the screen, containing one or more fields.

Fields are the basic components of forms and can vary in shape and size, depending on the number and format of the items inside. Some fields take up the whole window, others take up enough space for only a few characters.

There are four types of fields: *menu*, *list*, *edit*, and *text*. Each field type is described in detail under "Fields," later in this chapter. See Figure 2–1, also, for typical examples of each field type.

An item is a piece of text within a field. Normally, items appear one per line in a single column. Depending on the type of field, items in a field can be selected or edited, or if the field is for viewing only, neither selected nor edited.

An application can display one or more forms simultaneously. The topmost form is the *active* process, capable of receiving keyboard input. Under the CTAM window manager, a terminal screen can display one or more *windows* (applications) running simultaneously. As with forms, the topmost window is the process currently capable of receiving standard input. The hierarchy of DPL entities running under the CTAM window manager is illustrated in Figure 4–4.

In the simplest case, a terminal screen displays one window (with invisible borders), the window contains one form, and the form contains one field with one item displayed. The shell is similar to a window displaying a form that contains a single edit field (the prompt line).

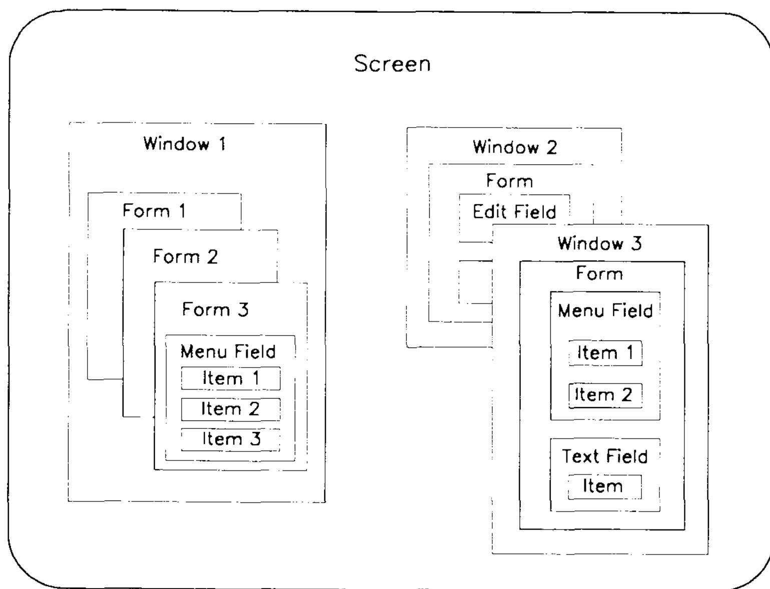

Figure 4-4. Hierarchy of DPL Entities Under CTWM

Forms

A complete forms definition file consists of a **form** statement and one or more **field** statements. A **form** statement is terminated by another **form** statement or by an end of file, and consists of the keyword **form** followed by a form name, an optional window label, and either the optional window coordinates or an optional placement keyword.

The first form to be displayed can be specified on the command line by using the **–s** option with the form name; for example:

```
$ dplrun -s firstform file.rf
```

where **file.rf** is the name of the DPL file, and **firstform** is the name of the starting form to be displayed. If the **-s** option is not used with a form name on the command line, the Dialogue Interpreter looks for a form named **mainform**.

The form name is also referred to by other forms in the DPL program, using the *action routine* called **PopupForm**, explained later in this chapter.

On the **form** statement, the window label is surrounded with quotes and follows the name. The coordinates of the upper left corner of the form are given in absolute window coordinates, with (1,1) being the upper left row and column, respectively, of the window in which the form appears. Optionally, the absolute coordinates of the lower right corner may also be given on **form** statements. Typical screens provide about 20 rows[1] and 80 columns. Here is an example **form** statement:

```
form formname "Form Label" (2,3)-(10,70)
```

As an alternative to the coordinates, the form designer can allow the forms system to decide on the placement of a form, based on one of three keywords:

```
form formname "Form Label" [son]
```

1. The bottom three lines of the screen are used for function key labels and prompts. Starting with a 25 line screen and subtracting three rows for labels and prompts, and two rows for the window border, there are 20 rows available for forms.

A **son** form is placed in a manner that partially but not completely overlaps the calling form, if possible. The **son** placement style is the default if no upper left hand corner is specified for the form.

```
form formname "Form Label" [new]
```

A **new** form is placed completely outside the calling form, and not over the top of any other existing forms, if possible.

```
form formname "Form Label" [popup]
```

A **popup** form is placed completely inside the calling form, if possible.

In the case where the form designer requires that the form label occupy the entire width of the form, the **fullwidth** flag can be used:

```
form formname "Form Label" [fullwidth] (5,10)-(10,60)
```

If a forms application is intended to be used in conjunction with the CTAM Window Manager, then the **resize** flag may be specified. The **resize** flag allows you to use the window manager[2] to change the size of a form.

Fields

A field is defined by the keyword **field** followed by a field name, a field type, the coordinates within the form where the field is to reside, and a comma-separated list of items; **field** statements are terminated with a semicolon. Only the field type and semicolon are actually required after the keyword **field**.

2. Window manager mode is accessible only when running with the CTAM Window Manager. For more information on the window manager, see **ctwm**(1W) in Appendix A.

A simple text field might be specified like this:

```
field FieldName text (5,10)
    "Press '$Enter' to continue.";
```

The field's coordinates (specified above as row 5, column 10) specify where within the form the field is to appear and are relative to the upper left hand corner of the form. In other words, if the above **field** statement is specified within a form that has its upper left corner at (2,2), then the upper left corner of the field is displayed 7 rows, 12 columns from the upper left corner of the window.

As with form coordinates, field coordinates are optional. In the following definition, things surrounded by square brackets may be omitted:

```
[ ([ top left row ] , [ top left col ]) [ - ([ bot right row ] , [ bot right col ]) ] ]
```

Elements of a field's coordinates that are not specified are figured out by the forms system at runtime, based on how many rows and columns are required to display the entire field. If the length of some items within a field may change at runtime, you can either specify a lower right hand coordinate large enough to accommodate the largest item in the field or *leave out* the lower right hand coordinate. For instance, if the variable **$Enter** is used in a text field, as in the previous example, the key name substituted at runtime for **$Enter** on some terminals might be **LineFeed** (eight characters long), while on other terminals the key name displayed might be **Go** (only two characters). By allowing the forms system to determine the lower right hand coordinate of the field, the display varies automatically for different length items in each field.

After the field type keyword (**menu**, **list**, **text**, or **edit**), a field *attribute* specification can be used. Field attributes can be used to alter some of a field's characteristics. Some attributes are useful only with some field types (each type is discussed in later sections of this chapter), while other attributes may be used with any field type.

Field attributes are specified with a comma separated list of attribute keywords enclosed by square brackets. Each field attribute is either on or off with the default depending on the field type. If an attribute keyword is preceded with a tilde (~), that attribute is turned off.

The following are general purpose field attributes:

boxed Causes a box to be drawn around a field. The box is drawn around the outside of the field's coordinates. If the field is vertically scrollable, a scroll bar is drawn along the right inside edge of the box. Likewise, if the field is horizontally scrollable, a scroll bar is drawn inside the box along the bottom edge. Default: off.

off If a field is off, the user is not able to move the cursor to the field. Normally, **text** fields are off; **menu**, **list**, and **edit** fields are on.

save When a form is closed and then later displayed, normally all of the field's values are recomputed and all defaults are reset. However, by specifying the **save** field attribute, a field's value is not recomputed but retains its state from the previous invocation. This is useful if you want a field's default value to be whatever the user entered previously.

vbar, hbar These flags permit scroll bars to be drawn for fields that do not have the **boxed** flag set. Default: off. Example:

```
field MyField menu [ boxed, ~vbar ] (2,2)
```

Menu Fields

A menu field description contains a list of selectable items to make up the field, and optional *action routines* to be executed if specified items are selected. Menu items with their action routines are separated by commas and terminated with a semicolon:

```
form formname "Form Label" (2,3)-(10,70)
    field MenuName menu (4,4)
        "Menu Item #1" onselect{ action routine #1 },
        "Menu Item #2" onselect{ action routine #2 },
        "Menu Item #3" onselect{ action routine #3 };
```

In this example, the user may select one menu item. When it is selected, the appropriate action routine is executed. (More information on action routines is given later in this chapter.)

The optional menu name (**MenuName** in the above example) is used in certain types of action routines to make use of the chosen menu item.

The coordinates (4,4) specify the upper left corner of the menu field, relative to the upper left corner of the form. The lower right corner may be specified by including a second set of coordinates as on the **form** statement.

To make a multiple column menu (where menu items appear as a table, in rows and columns), an asterisk is used following the relative coordinates. Example:

```
form formname "Form Label" (2,3)-(10,70)
     field MenuName menu (4,4) * 3
          "Item 1",
          "Item 2",
          "Item 3",
            .
            .
            .
          "Item 50";
```

The "* 3" makes this a three column menu. Depending on the form and menu size, the field may automatically be made to scroll either horizontally or vertically.

The example below demonstrates a multiple column menu for which the number of rows and columns is decided at runtime by the system. A best fit is made:

```
field name menu (2,2)-(,70) *
```

This example demonstrates a multiple column menu that occupies 20 rows, and as many columns as is necessary:

```
field name menu (2,2)-(22,)*
```

Optional menu field attributes appear following the **menu** keyword, in a comma-separated list of keywords inside square brackets. These attributes affect the entire menu. To turn on the attribute, place it in the list. To turn it off, prefix a tilde (~). In this example, the selected item is check marked, not highlighted, and the entire menu field of the form is surrounded by a box:

```
form example "Example" (2,3)
    field menu [ ~high, chcksel, boxed ] (2,2)
        "Menu Item #1" onselect{ action routine #1 },
        "Menu Item #2" onselect{ action routine #2 },
        "Menu Item #3" onselect{ action routine #3 };
```

Here is the full list of menu field attributes:

high This uses a highlighted bar as the cursor. Default: on.

chck This uses a check mark as the cursor. Default: off.

highsel This leaves the highlighted bar on the selected item(s). Default: off.

chcksel This leaves a check mark on the selected menu item(s). Default: off for menus, on for lists.

dash This causes a leading dash to be printed on the current item. Default: on.

pulldown If the user presses the **Command** key (usually **Control-C**), the menu expands to display all of the items.

writein This allows the user to type his or her own item instead of choosing one from the menu. The typed in value becomes the field's value instead of the item under the cursor.

List Fields

A list field is actually a menu field in which more than one item may be selected at a time. For example:

```
form PrintForm "Select Files To Be Printed" (4,4)
  field items list (2,2)
    "File1" onselect{ action routine #1 },
    "File2" onselect{ action routine #2 },
    "File3" onselect{ action routine #3 };
```

A facility exists to use the multiple values that would exist in **$PrintForm.items** in the above example. A named list field becomes an array, like in the C programming language, containing the values that the user entered. The count, useful on the **while** command,

documented in a later section, is in element [0] of the array. For example:

```
form PrintForm "Select ONLY THREE files to be printed" (4,4)
     onselect {
          doexec( "lp", $PrintForm.Items[1] );
          doexec( "lp", $PrintForm.Items[2] );
          doexec( "lp", $PrintForm.Items[3] );
     }
     field Items list (2,2)
          'ls';
```

In the example above, note that a shell command surrounded by backquotes is used to determine the field's elements at runtime. For more information on defining menu items at runtime, see "Items," later in this chapter.

Edit Fields

An edit field defines an area on the screen into which the user types characters. Edit fields can be blank filled or they can be initialized either statically with a string constant or dynamically with a shell command. This edit field is blank filled:

```
field fieldname edit (2,2)-(2,50) "";
```

This edit field is initialized with a string constant:

```
field fieldname edit (2,2)-(2,50) "Default Value";
```

This edit field is initialized with the output of a shell command:

```
field fieldname edit (2,2)-(2,50)
     'grep MyName /etc/passwd | cut -d: -f5';
```

If the **save** attribute is used on the edit field, and the form is displayed for a second time during a session, the edit field is initialized with the value that the user entered on the most recent use of the form. Shell commands, as in the previous example, are not run a second time. Example:

```
field fieldname edit [ save ] (2,2)-(2,50)
     "First Default Value";
```

The **blank** attribute indicates that the edit field is silent; that is, when a value is entered into it, the value is not echoed to the screen. **Blank** is the only attribute that is entered *after* the double quotes in a **field** statement. Example:

```
field text (2,2) "Enter your password: ";
field fieldname edit (2,23)-(2,40) "" [blank] ;
```

Edit fields may span more than one line. Example:

```
field fieldname edit (2,2)
    "name",
    "address",
    "phone number";
```

The value resulting from a multiline edit field is a string containing newlines where the line boundaries are.

The value of an edit field may be type checked. The initial value of the edit field is followed by a colon (:) and then a *type verification string*. Example:

```
field fieldname edit (2,2)-(2,4)
    "" : "###";
```

In this example, the forms system causes the terminal to beep if the user enters anything other than 3 numeric digits.

The following characters can be used in a type verification string:

- **Pound Sign**. The # character means digits. That is, the characters 0–9, as in the example above.
- **Question Mark**. The ? character stands for letters. These include A–Z, a–z, and possibly other characters, depending upon the national character set currently in use. For example:

  ```
  field description edit (1,1)-(1,15)
      "" : "????????????????";
  ```

  This requires the user to enter a sixteen character name, using only letters. Note also that in this case, the user would be required to fill in all sixteen blanks with a letter.

- **Period**. This is the wild card string. It matches anything that the user enters. For example:

```
field PartNumber edit (1,1)-(1,9)
    "" : "####.?????";
```

  This example requires the user to enter a four digit part number, followed by any character (the wild card) followed by five letters.

- **Brackets**. Brackets define the ranges of type verification letters. In the example below, the user must enter a number in the range 0–5, followed by a letter in the range n–z:

```
field value edit (1,1)-(1,2)
    "" : "[0–5][n–z]";
```

  A leading caret (^), used within bracketed ranges, allows the programmer to require any character *but* the range specified. In the example below, the operator may enter anything but numeric digits:

```
field value edit (1,1)-(1,5)
    "" : "[^0–9]";
```

- **Any character**. By placing a character or set of characters in a type verification string, the user is required to enter this character value as part of the edit field. In the example below, the 'dash' character (–) is required:

```
field PhoneNumber edit (1,1)-(1,11)
    "" : "###–###–####";
```

  This example requires the operator to enter a phone number. He must enter the dashes with the appropriate number of digits.

As shown in the examples above, type verification strings may be combined to build more complex input strings. In addition, the following constructs may be used along with these strings to aid the programmer in developing more complex verification strings:

- **Asterisk**. When an asterisk follows a type verification string, the forms system interprets it as meaning "zero or more of" that string. Example:

```
field number edit (1,1)-(1,5)
      "" : "#*";
```

  This means that the operator must enter zero or more digits into that edit field.

- **Plus**. When a plus sign follows a type verification string, the forms system interprets it as meaning "one or more of" that string. Example:

```
field name edit (1,1)-(1,5)
      "" : "?+";
```

  This means that the operator must enter one or more letters into that edit field.

- **Braces**. A range of occurrences may be specified with the use of braces. Example:

```
field name edit (1,1)-(1,6)
      "" : "#{1,3}?*";
```

  This means that the operator must enter between one and three digits optionally followed by any number of alphabetic characters (until the field is full of course).

NOTE

It is advisable to keep edit type specifications as simple as possible; otherwise, the user may be placed in an awkward situation in which *no key is valid*. For instance:

```
field name edit (1,1)-(1,10)
      "" : "#+????";
```

In this example, the user is allowed to type any number of digits but must still type four alphabetics. If the user enters a large number, the field can never be valid, because there will not be room left to type the four alphabetics.

For additional field verification constructs, listen to me now and read about it later but know this next week: the **onvalid** and **return** features are described under "Event Language Control Flow," later in this chapter.

Text Fields

Text fields cannot be edited or selected; they are used as labels for other fields, or they provide additional information in a form. Text fields may be sets of quoted strings separated by commas, they may end in a semicolon, or they may be assigned by the means of a shell command. If the amount of text to be displayed exceeds the size of the form, the text is made scrollable. Here is an example of a text field being used as a prompt for an edit field:

```
form mainform
     field text (2,2) "Enter your name: ";
     field name edit (2,19)-(2,50) "";
```

The **tail** field attribute flag may be used with a text field to cause a scrollable text field to be initially bottom justified.

Items

An item is a piece of text within a field. Normally, items appear one per line in a single column. Depending on the type of field, items in a field can be selected or edited, or if the field is for viewing only, neither selected nor edited.

If there are more items in a field than will fit within the coordinates specified for the field, the field is made *scrollable*. Such fields are conventionally specified with the **boxed** field attribute to make their special property more apparent to the user.

Most of the examples of menus up to this point have contained predefined menu items, that is, the form designer has coded the selections directly into the form, such as *file1*, *file2*, and so on. This is impractical in certain types of applications. To get around this problem, the Dialogue Interpreter supports backquotes similar to the way the shell supports them; in this case, executing shell commands to get menu

items. The output of the shell command is captured by the forms system, and used for the menu items. For example:

```
form RmForm "Remove A File" (2,3)-(20,60)
    onselect{ doexec( "rm", $RmForm.filename); }
    field filename menu (2,2)-(8,20)
        'ls | sort';
```

In this example, the menuing system runs the **ls** command, pipes the result through **sort**, and uses the resulting file list as the menu item list. Thus, the menu items are not coded directly into the form, but are built when the form runs and are comprised of file names from the current directory.

Item Attributes

Items can be defined with an *item attribute*, following the quoted string (before the equals sign, if any) and surrounded with square brackets. These attributes affect only the item with which they appear.

root This attribute instructs the Dialogue Interpreter to display the item only if the user is root. Typically, this attribute is used in system administration routines.

level This attribute works much like **root**, except that it gives more complete control to the application designer in terms of who can use the given menu items. There are eight levels of end users: zero through seven. Level zero has the highest amount of capability, and is equal to root. The next levels can be used to gradually give users more capabilities. Example:

```
form startform "Things you can do" (2,2)
    field menu (2,2)
        "All Capabilities" [level=0],
        "System Programmer Capabilities" [level=1],
        "Application Administrator" [level=2],
        "Data Entry Clerk" [level=3];
```

The **level** value is known to the dialogue interpreter by looking at the *effective group ID value*.[3] Members of group zero (which is root's group) are set to level zero. Members of group 100 are set to level 1; group 101 to level 2, group 102 to level 3, and so on. End users see menu items marked with their level number and above. Note that programmers building menus into their C programs (discussed in Chapter 5) can set the level by means of the **SetLevel** subroutine call and can get the level by means of the **GetLevel** call.

default Instructs the Dialogue Interpreter to regard that item as the default choice, and to position the cursor on that item when the field is displayed. Example:

```
form RmForm "Remove A Subtree" (2,3)
    onselect{ doexec( "rm", "-r", $RmForm.filename); }
    field filename menu (2,2)
        "/u/YourHomeDir",
        "/tmp" [default],
        "/usr/misc" [root];
```

high The item is to be displayed in reverse video at all times.

blank The item is to be blank (all spaces). This is useful in **edit** fields when it is desirable to inhibit echoing to the screen for things such as passwords.

Item Values

The value of an item is usually the same as what appears on the screen. However, a facility exists to substitute an alternate item value that is different from the value that the user sees on the screen. That is, an item can have a *user value* and a *display value*. For the example below, assume that *file1* and *file2*, and so on, are file names. In this example, the file names are not what the user sees as menu

3. A user's *effective group ID* is usually the same as the user's group ID field in the **/etc/passwd** file. Detailed discussions of how a user's effective group ID is determined are contained in Section 2 of the *CTIX Operating System Manual*.

items. Instead the user sees menu items labeled *The First File*, *The Second File*, and so on.

```
form CatForm "Cat A File" (2,3)
     onselect{ doexec( "cat", $CatForm.filename); }
     field filename menu[ high ] (2,2)
          "The First File"= "file1",
          "The Second File"= "file2",
          "The Third File"= "file3",
          "The Fourth File"= "file4";
```

The equals sign is followed by a quoted string, which is the value that is placed into the variable **$CatForm.filename** when the menu item is selected.

The value that is displayed is called the *display value*. The value that is used in the program is called the *user value*. For information about the way programs can distinguish between display and user values, see "Variables," later in this chapter.

Events

An *event* defines the action or actions to be performed when the selection associated with a form, field, or single menu item is complete. Curly brackets {} surround the action or actions to be performed. Listed below are descriptions of each DPL event:

onact This event is executed each time the field is activated, meaning each time the field is entered with the **Tab** key.

onbadkey This event is executed when the user types a key that is not valid in the current context; for instance, if the user types the **BackSpace** key in a menu field, or types a key that is considered invalid for a type verified edit field.

oncancel This event is activated by the **Cancel** key.

onclose This event is activated by the **Finish** key.

onhelp This event is activated by the **Help** key.

oninit This event is executed when a form is first displayed by the forms system.

onkey This event is executed when any keystroke is entered; it can also be configured to execute when a specific key is entered.

onselect This event is activated by the **Enter** key.

onvalid This event is executed when the user attempts to exit a field. The programmer can perform value verification in the **onvalid** event.

The DPL programmer has the use of many of the features of a traditional programming language in building event code. Action routines such as **doexec**, **dohelp**, or **PopupForm** take on the form of subroutine calls, while keywords such as **if, else, while,** and **return** offer flow of control, allowing more useful programs to be written. The event language also supports variables. For more information, see "Event Language Control Flow" and "Variables," later in this chapter.

In most of the preceding examples, there has been a one-to-one correspondence between a menu item and an action taken when that item is selected. This type of usage is frequently seen when a different kind of action is performed based on which menu item is selected.

In other types of menu sessions, the same kind of action is performed regardless of the menu item selected. For these sessions, the menu item acts as data used in the action that is performed.

In the following example, the **doexec** action routine executes the program whose name is given, and passes it any parameters provided. In this case, the result is to **cat**(1) one of four files:

```
form CatForm "Cat A File" (2,3)
     field menu[ high ] (2,2)
          "file1" onselect{ doexec( "cat", "file1"); },
          "file2" onselect{ doexec( "cat", "file2"); },
          "file3" onselect{ doexec( "cat", "file3"); },
          "file4" onselect{ doexec( "cat", "file4"); };
```

Using a single **onselect** command and by naming the menu, this example shows a different way to cat one of the files:

```
form CatForm "Cat A File" (2,3)
     onselect{ doexec( "cat", $CatForm.filename); }
     field filename menu[ high ] (2,2)
          "file1",
          "file2",
          "file3",
          "file4";
```

In this example, the **onselect** statement is positioned outside of any field on the form. When used in this way, the **doexec** action routine is executed regardless of which menu item is selected. (For more information on the placement of the **onselect** statement, see "Events," later in this chapter.)

The form name **CatForm** is used in the example to define a structure with a field using the menu name **filename**. Again, these are used much like C structs; in this case using the name of the form, a dot, and the name of the field.

The use of the dollar sign is much like that in the shell language; it tells the Dialogue Interpreter to substitute the actual value of the variable.

Scoping of Events

Each event can be associated with a form, a field, or an item. For example, the **onhelp** event below is associated with a form. This means that when the user presses the **Help** key, the same help message is displayed, no matter which field of the form the cursor is in at the time:

```
form mainform
     onhelp{ PopupForm( helpform ); }
     field field1 menu
          .
          .
          .
     field field2 edit
          .
          .
          .
```

In the following example, the **onhelp** events are associated with fields. This means that when the **Help** key is pressed, if the cursor is in *field1*, one help message is displayed, and if the cursor is in *field2*, a different help message is displayed. Example:

```
form mainform
     field field1 menu
          onhelp{ PopupForm( helpform1 ); }
          "Item1",
            .
            .
            .
     field field2 edit
          onhelp{ PopupForm( helpform2 ); }
          "Default Value";
            .
            .
            .
```

Note that when an event is associated with a field, the event appears before the items in that field. When an event is associated with a particular item in a field, the event is activated only if that item is selected. In the example below, the event follows the menu item:

```
form mainform
     field menu
          "Item1" onselect { doexec( "prog" ); },
          "Item2" onselect { doexec( "prog2"); };
```

The Onkey Event

As previously described, the **onkey** event is executed when any keystroke is entered, or it can be configured to execute when a specific key is entered. This example traps on every keystroke:

```
form mainform "Trap All Keys" (2,2)
     onkey{ NoteForm( "Only arrow keys and '$Enter' are legal");}
     field menu "Item1", "Item2", "Item3";
            .
            .
            .
```

Like all events, the **onkey** event can be scoped to respond to all fields, a single field, or a single item.

A special type of **onkey** event allows the forms programmer to trap only certain keys:

```
form mainform "Trap the $Cmd key" (2,2)
     field edit "" onkey($Cmd){ PopupForm( helpform ); };
                    .
                    .
                    .
```

In this example, if the user presses the **$Cmd** key (such as **Control–C** on a VT100 terminal), the **PopupForm** action routine is executed. The **onkey** event appears with a parameter surrounded by parentheses; the parameter represents the name of a key, and it can be: **$Cmd** for the command key, **$F1** through **$F10** for the function keys, and **$RollUp** or **$RollDn** for the scrolling keys.

The key names that are available are listed in the file **/usr/include/kcodes.h**. To use **onkey** for any special keys, such as the function keys or arrow keys, get the name of the key from this file and put a dollar sign before it. To perform keystroke handling on a standard ASCII key, surround it with single quotes just like a C character constant in the **onkey** statement:

```
onkey( '!' ) { doexec( "/bin/sh" ); }
```

To trap more than one key, use multiple **onkey** statements.

Action Routines

The **doexec** action routine has the syntax of a C subroutine call. Up to 32 parameters become the name and command line values of an "exec'ed" program:

```
onselect { doexec( "ls", "*.rf" ); }
```

More than one action routine can appear on an **onselect** event, separated by semicolons.

```
onselect { doexec( "ls", "file1" );
      doexec( "sort", "file1" ); }
```

The **PopupForm** action routine specifies the form name and the parameters to be passed to the form. The parameters are quoted strings, variables, or output from shell scripts. They correspond to **$1, $2, $3**, and so on, in the called form. Up to thirty parameters are allowed. End the list of parameters with a zero. Example:

```
form mainform "Demonstrate PopupForm Parameters" (2,3)
     onselect{ PopupForm( otherform, "Item1", "Item2",
                    `echo Item3`, $mainform.Name, 0);}
     field Name edit (1,1)-(1,30) "Item4";

form otherform "Uses the Items from mainform" (6,8)
     field menu (2,2)
          "$1",
          "$2",
          "$3",
          "$4";
```

When the above example is run, the value of **$1** is *Item1*, **$2** is *Item2*, **$3** is *Item3*, and **$4** is the value that the user entered for the field *Name* in *mainform*.

The first parameter in **PopupForm** may be either the name of a form, or a string that evaluates to the name of a form. Example:

```
form myform
     onselect{ PopupForm( $myform.mymenu ); }
     field mymenu menu
          "Choice1" = "otherform1",
          "Choice2" = "otherform2",
          "Choice3" = "otherform3";

form otherform1
     .
     .
     .
form otherform2
     .
     .
     .
form otherform3
     .
     .
     .
```

The **NoteForm** action routine pops up a standard form labeled *Note* and displays the text provided:

```
form mainform "Tests Noteform" (2,2)
     onselect{  NoteForm( "This is a text message" ); }
     field ...
       .
       .
       .
```

The forms system waits for the user to press the **Cancel** or **Finish** key before clearing the form from the screen and continuing.

The **ErrorForm** action routine works like **NoteForm**, but the form is labeled *Error*.

The **SetPrompt** action routine specifies the value of the prompt line that is displayed when a field is active:

```
form mainform "Set up user information" (2,2)
     field loginid edit
          oninit{ SetPrompt( "Enter login ID" );}
               "" ;
     field name  edit
          oninit{ SetPrompt( "Enter your name");}
               "" ;
     field shell menu
          oninit{ SetPrompt( "Pick a shell" );}
          "C-shell",
          "Bourne shell";
```

In the example above, the prompts are associated with edit fields and menu fields. They are displayed at the bottom of the screen when the field is entered. To associate a prompt with a text field, turn the field on by using the field attribute [~**off**].

The **SetRefreshRate** action routine allows the programmer to refresh the information on the screen after a specified interval. It is especially useful for administrative programs that display the status of some part of the system. Example:

```
form mainform "Show the current time" (2,2)
     field text (2,2) oninit{ SetRefreshRate( 5 ); }
          'date';
     field text (4,2)
          "Press '$Cancl' to get out of this.";
```

In this example, the forms system refreshes the text field every five seconds, in this case re–initializing the field by running the **date** program. This means that the time would be updated every five seconds.

The **RefreshField** action routine is used to repaint the values in a field without redrawing the entire form. Example:

```
form mainform "Dynamic Menu Items" (2,2)
    oninit{
        $ItemList = "Item1 Item2 Item3";
        set $ItemList;
    }
    onselect{
        $ItemList = $ItemList + "NewItem";
        set $ItemList;
        RefreshField( "MyMenu" );
    }
    field MyMenu menu (2,2)
        "$1","$2","$3","$4","$5","$6","$7","$8";
```

In this example, the variable **$ItemList** is assigned a value using the command **set**. Variables can be set anywhere within event code.

If **RefreshField** is used without a field name, the currently selected field is refreshed.

The **LabelKey** action routine allows the programmer to put a six character label for the function keys at the bottom of the screen. Up to 10 function keys can be displayed. The following example uses a **while** statement (see "Event Language Control Flow," in this chapter) to label all ten keys:

```
form mainform "Label function keys" (2,2)
    oninit {
        $i = 1;
        while( $i <= 10 ) {
            LabelKey( $i, "Key #"+$i );
            $i = $i + 1;
        }
    }
    onkey( $F1 ) { PopupForm( f1form );
            LabelKey( 1, "Done" ); }
    onkey( $F2 ) { PopupForm( f2form );
            LabelKey( 2, "Done" ); }
    onkey( $F3 ) { PopupForm( f3form );
            LabelKey( 3, "Done" ); }
                .
                .
                .
```

The first parameter is the key number (1–10, numbered left to right), and the second is the key label value.

The **LabelForm** action routine changes the label at the top of the form. Example:

```
form mainform "Original Label" (2,2)
     onselect{
          LabelForm( "New Label" );
           .
           .
           .
```

The **SetSelect** action routine dynamically sets the default item in a list or menu field. It is most useful in applications that build the menu items at runtime. Example:

```
form mainform "Pick a file" (2,2)
     field menu [chcksel] (2,2)
          oninit{ SetSelect( "UsersFile" ); }
          'ls';
```

In this example, the menu is built at runtime by running **ls**. The **SetSelect** action routine causes the file *UsersFile* to be marked as the default.

GotoForm works just like **PopupForm**, transferring control to a new form. It differs in that it closes the current form, and when the new form is exited, control returns to the original parent form. Example:

```
form form1 "demo"
     onselect{ PopupForm( form2 );}
     field text "Press '$Enter'.";
form form2
     onselect{ GotoForm( form3 );}
     field text "Press '$Enter'.";
form form3
     field text "When you exit, control returns to form1";
```

The currently active field can be controlled with the action routines **AdvanceField**, **BackField**, and **SetCurrentField**. **AdvanceField** and **Backfield** cause the form to behave exactly as if the user had typed the **Tab** or **BackTab** keys. **SetCurrentField** takes a single argument that is the name of a field in the current form; that field becomes the active field.

The **CloseForm** action routine closes the current form as if the user had typed the **$Close** key (**Finish** on a PT, '^D' on most other terminals). Statements following **CloseForm** continue to be executed until there are either no more statements or the **return** statement is

executed. Since **CloseForm** requires no parameters but is an action routine, it requires empty parentheses to follow it. Example:

```
form MyForm "Illustrate CloseForm()" (2,2)
     onselect{
          PopupForm( form1 );
          PopupForm( form2 );
          CloseForm();
     }
            .
            .
            .
```

Event Language Control Flow

The keywords **if, else, while,** and **return** can be used to make up the code in an event along with action routines, such as **PopupForm, doexec**, and **dohelp**, described in the previous section. The syntax for the control flow statements is much like that of the C programming language. Example:

```
form mainform "Choose One Item" (3,3)
     onselect{
          if ( $mainform.MyField == "Item 1" )
               PopupForm( form1 );
          else
               PopupForm( form2 );
     }
     field MyField menu (2,2)
          "Item 1",
          "Item 2",
          "Item 3";
```

In the above example, control is transferred to **form1** if the menu item chosen is *Item 1*; otherwise, control is transferred to **form2**. In this way, flow of control is routed either to one **PopupForm** call or another.

Loops can be built using the **while** keyword:

```
form mainform "Build Your Data Set" (2,2)
     onselect{
          $done = "Continue";
          while( $done == "Continue" ) {
               PopupForm( CreateFile );
               PopupForm( AskIfDone );
               $done = $AskIfDone.answer;
          }
          PopupForm( IntegrateDataSets );
          CloseForm();
     }
     field name edit (2,19)-(2,50) "";
     field text     (2,2) "Enter your name: ";

form AskIfDone "Are you done?" (4,4)
     field answer menu (2,2)
          "Continue",
          "All done";

form CreateFile "Enter the file name" (4,4)
     field name edit (2,2)-(2,20)
          onselect {
               'cp /etc/passwd $CreateFile.name';
               CloseForm();
               }
          "";
```

The above example asks for the user's name once, and then loops, creating files, until the user selects the *All done* menu item in the form **AskIfDone**.

In addition, this example illustrates the use of *variables*, which are described in the next section. In brief, the variable **$done** can be used anywhere within the event in the example above. The value of **$done** could be a string value assigned from constants (such as *Continue*), a field value (such as **$AskIfDone.answer**), the result of a shell script (such as, **grep Gary /etc/passwd | cut -d: -f6**), or the value assigned from another variable.

Values are assigned to variables using the equals sign (=) operator in statements that end with a semicolon:

```
$variable = "Value";
```

The operands that can appear in **if** and **while** statements are the same as the values that can be assigned to variables: field values (such as, **$MyForm.MyField**), string constants (values inside quotes), variables (**$MyVar**), the output from shell scripts (such as, **echo xyz**), integer

constants, character constants, and action routines (used to check the error return value).

The following operators can be used to make comparisons within **if** and **while** statements:

```
==      Equal to
!=      Not equal to
<       Less than
<=      Less than or equal to
>       Greater than
>=      Greater than or equal to
||      OR
&&      AND
(       Left parenthesis
)       Right parenthesis
!       NOT
```

Example:

```
form mainform "Choose a menu item" (2,2)
    onselect{
        $x = $mainform.item;
        if ( $x == "Item1" || $x == "Item2" )
            PopupForm( demo );
    }
    field item menu (2,2)
        "Item1",
        "Item2",
        "Item3";
```

In this example, the form **demo** is displayed if the menu item chosen is *Item1* or *Item2*.

The following example illustrates the use of error returns from action routines:

```
form mainform "See if $Cancl was hit" (2,2)
    onselect{
        if ( PopupForm( MyForm ) != 0 )
            NoteForm( "You pressed $Cancl, didn't you?" );
        else
            NoteForm( "MyForm was exited normally" );
        }
            .
            .
            .
```

If an **if** or **while** statement contains a string of comparisons using AND, the interpreter performs them one after the other from the left to the right. If any one of them fails, it does not perform the

remaining comparisons, since the entire **if** would not evaluate to TRUE. This has an interesting spinoff, when used with action routines:

```
form mainform "Calls lots of forms" (2,2)
     onselect{
          if( PopupForm(f1) == 0 && PopupForm(f2) == 0 )
               PopupForm( f3 );
     }
               .
               .
               .
```

In this example, the interpreter first displays form **f1**. If the user presses the **Cancel** key, the return value of the **PopupForm(f1)** call becomes nonzero. In this case, the interpreter does not even make the call to **PopupForm(f2)**, since the entire **if** statement could never evaluate to TRUE.

In cases where there is more than one statement to be executed after an **if** or an **else**, curly brackets { } are used to group the statements together:

```
form frm "If test" (2,2)
     onselect{
          if ( $frm.f1 == "Item1" && $frm.f2 == "Item2" ) {
               PopupForm( FirstTrueForm );
               PopupForm( SecondTrueForm );
          }
          else {
               PopupForm( FirstFalseForm );
               PopupForm( SecondFalseForm );
          }
     field f1 menu "Item1", "Item2", "Item3";
     field f2 menu "Item1", "Item2", "Item3";
```

The **return** statement is used to exit the event. The application user sees the same form remaining on the screen. He then has the opportunity to select other items and press **Enter** again, or to exit the form using **Finish** or **Cancel**. Example:

```
form MyForm "Illustrate return" (2,2)
     onselect {
          if ( $MyForm.f1 == "Bad Choice" ) {
               NoteForm( "Rotten choice.  Try again" );
               return;
          }
          PopupForm( nextform, $MyForm.f1 );
     }
     field f1 menu
          .
          .
          .
```

When used with the **onvalid** event, the **return** statement can use a parameter, such as **$TRUE** or **$FALSE**, which controls the handling of invalid values. If the return statement has a **$TRUE** parameter value, the operator may enter additional field values or exit the form. If the return statement has a **$FALSE** parameter value, the user may not exit the current field. Instead, he must enter a value that satisfies the constraints of the **onvalid** event. Example:

```
form mainform
     field text (2,2) "Enter your age: ";
     field age edit (2,18)-(2,20)
          onvalid {
               if($mainform.age > 100 || $mainform.age == 0) {
                    NoteForm( "Bad value.  Try again." );
                    return $FALSE;
               }
               return $TRUE;
          }
          "  " : "#+";
```

In this example, the type verification string ("#+") appears with the edit field. This assures that the user enters only digits, but does not assure that he enters a number that is useful to the application. The **onvalid** event allows the value to be checked, in this case using an **if** statement, and returns the user to the edit field if the value is not within a given range.

Variables

Variables are set using the equals sign (=) operator. Wherever a variable appears in the program, its assigned value is substituted by the form system. A variable is *global*; that is, its value may be accessed and set anywhere that it appears. To access its value, the

variable should appear with no quotes around it, unless it is used outside of the event. For example, if a variable is used as a menu item or as an edit field initial value, it should have quotes around it:

```
form mainform "Demonstrate Variables" (2,2)
     onselect{
          $value = "Item2";
          PopupForm( otherform );
          CloseForm();
     }
           .
           .
           .
form otherform
     field menu
          "Item1",
          "$value",
          "Item3";
           .
           .
           .
```

In the event following the **onselect** statement above, the variable **$value** appears without quotes; when it appears in **otherform**, it is surrounded by quotes, as are all menu items.

Generally, the value of a variable is a string value. The plus operator can be used to concatenate string values together:

```
$value = "String1" + "String2";
```

The resulting value is "String1String2". Another example:

```
$x = "X" + $MyForm.MyField + 'echo xxx' + $1 + $variable ;
```

Variables may assume numeric values, and arithmetic may be performed on those values. The forms system notes the type of value being assigned to a variable and performs integer operations at appropriate times.

```
form MyForm "Enter the correct value" (2,2)
     onselect{
          $tries = 0;
          while ( $tries != 3 ) {
               PopupForm( Question );
               if ( $Question.Answer == "Bad Answer" ) {
                    NoteForm( "Try again." );
                    $tries = $tries + 1;
               } else {
                    PopupForm( cleanup );
                    CloseForm();
               }
          }
          return;
     }
            .
            .
            .
```

In the above example, the forms system notes the type of value assigned to the variable **$tries**, in this case a numeric value, and stores that value so that numeric operations can be performed on it. Thus, when the forms system encounters the plus operator, it performs an addition of numerics rather than a concatenation of alphabetics.

The legal operations for numeric variables are:

```
+     addition
–     subtraction
*     multiplication
/     division
%     modulo
```

Only the addition operation is valid for both strings and numeric variables. In the case where a string and a numeric appear in the same operation, the forms system converts the numeric to a string:

```
$x = 123 + "xyz";
```

The resulting value would be the string "123xyz". If a numeric variable is assigned a string value, the forms system converts the type of the variable to be string. The result below would be "579string":

```
$x = 123 + 456;
$x = $x + "string";
```

The forms system converts operands into strings to perform comparisons. As a side effect, integer comparisons of == and != work, but inequalities do not. The **MakeInt** routine forces integer comparisons. For example:

```
                .
                .
                .
onselect {
  $i = 12;
  $j = -13;
if ( MakeInt ($i) <= MakeInt ($j) )
{
                .
                .
                .
```

Here is an example of arrays, using numeric variables, where the value comes from a **list**:

```
form mainform "Print some files" (2,2)
      onselect{
            $count = $mainform.files[0];
            $i = 1;
            while( $count != 0 ) {
                  doexec( "lp", $mainform.files[$i] );
                  $i = $i + 1;
                  $count = $count - 1;
            }
            CloseForm();
      }
      field files list (2,2)
            'ls';
```

Recall that the number of list items selected by the user is available in element [0] of the array. Its value is used as a loop variable, to control the number of iterations through the loop.

Variable names that represent fields in the form may be manipulated in the same way that any other variables are manipulated. For example, they can be assigned like this:

```
form mainform
      onselect{
            if ( $mainform.field1 != "Special Case" )
                  $mainform.field2 = "Special Choice";
              .
              .
              .
```

By default, **$mainform.field1** contains the *user value* of the field *field1* in the form *mainform* when referenced in an expression. However, when assigned in the manner shown above, **$mainform.field1** refers to the *display value* of the field. A convention exists that allows either of the suffixes **.dval** and **.uval** to be added to a field name to override the default values, if needed. The **.dval** suffix explicitly refers to the display value of a field, the **.uval** suffix to the user value of a field. Thus, the two assignments below are equivalent:

```
$MyForm.MyField = "value";
```

and

```
$MyForm.MyField.dval = "value";
```

When a value is assigned to a visible field, the field's value on the screen changes. To change both the user value and the display value, use the **$ITEMFS** reserved variable as shown, and separate the two values with an equals sign:

```
form mainform
    onit {
        $ITEMFS = "=";
        $X = "The first file=file1\nThe second file=file2\n
             The third file=file3";
    }
    field menu
        "$X";
```

Defining Global Variables

Variables may be defined in a **.rf** file by placing assignment statements before the first form definition. This allows *global variables* to be defined before any particular action routine is executed. For example:

```
$message1 = "Loading in files...";
$message2 = "Unable to find file $1";
    .
    .
    .
form mainform
    .
    .
    .
```

Using this mechanism, a program may increase its nationalizability by placing all of its text in a **.rf** file.

Special Variables

There are a number of *special variables*, in addition to the key name variables, such as **$Enter, $Exit, $Cancl**, and **$Help**. The **$Cmd** variable has the name of the key that the user presses to get the current command options. This is, for example, **Control-C** on many standard terminals. **$Cmd** is most logically used with the **onkey** event. Example:

```
form mainform "Enter a login name" (2,2)
     field myfield edit (2,2)-(2,30)
          onkey( Cmd ) {
               PopupForm( namelist );
               $mainform.myfield = $namelist.choice;
          }
          "$LOGNAME";
     field text "Enter '$Cmd' for a list of choices";

form namelist "Choices of login names" (5,5)
     field choice menu "fred", "joe", "mary", "bob";
```

The **$KEY** special variable contains the value of the most recent key entered by the operator. This value can be obtained by viewing the file **/usr/include/kcodes.h**.

The **$ERROR** special variable is used to hold the error return value from the most recent shell script. Example:

```
form mainform "Create A File" (2,2)
     onselect{
          `test -f $mainform.name `;
          if ( $ERROR != 0 )
               PopupForm( CreateFile, $mainform.name );
          else
               ErrorForm( "File already exists" );
     }
     field name edit (2,2)-(2,30) "";
```

In the example above, the user enters the name of a file. If the file does not exist, the error return from the shell **test** command is non–zero, allowing the **PopupForm** call to display the form **CreateFile**, which presumably creates the file. Otherwise, an error message is displayed.

Recall that the error return from a **doexec** command is assigned like this:

```
$ThisError = doexec( "cat", $mainform.name );
```

The forms system allows access to the user's environment variables, such as **$LOGNAME**, **$SHELL**, and **$HOME**. When the application program starts running, the forms system reads in the values of the user's environment variables, and creates variables with those names. If the form program references an environment variable that isn't defined for the user, then the forms system assigns a zero length string to that variable.

The **$LOGNAME** environment variable contains the login name of the current user:

```
form mainform "Use the user's name" (3,3)
     onselect{
          NoteForm( "Hello there, "+$LOGNAME );
               .
               .
               .
```

Note that the concatenation operator (plus sign) has been used to build a single string value that is passed to **NoteForm**.

The **$HOME** environment variable contains the home directory for the current user:

```
form mainform "Remove A File" (2,2)
     onselect{
          doexec( "rm", $HOME+$mainform.filename );
     }
     field filename edit (2,2)-(2,20) "";
```

The **$SHELL** environment variable contains the full pathname of the default shell for the current user:

```
form mainform "Select an action" (2,2)
     field menu (2,2)
          "Shell Escape"   onselect{ doexec( $SHELL ); },
          "Run the editor" onselect{ doexec( "vi" ); },
            .
            .
            .
```

The **$LEVEL** special variable contains the value of the *level* at which the program is running.

Summary of Terms

This section summarizes most of the terms introduced in this chapter.

- *Field Attributes* affect an entire field. Keywords: **boxed, high, chck, chcksel, highsel, tail, save, off, vbar, hbar, writein, pull-down**, and **dash**.
- *Item Attributes* affect only the menu item or list item on which they appear. Keywords: **root, level, blank, default**.
- *Events*, which are entered when the user presses a selection key such as **Finish, Enter**, or **Help**, include the following: **onselect, onhelp, onkey, oninit, onclose, oncancel, onact, onvalid, onbad-key**.
- *Action Routines* take the form of subroutine calls that are used following **onselect** statements: **PopupForm, NoteForm, ErrorForm, doexec, SetPrompt, CloseForm, SetRefreshRate, LabelForm, LabelKey, RefreshField, SetSelect, GotoForm, dohelp, SetCurrentField, GetCurrentField, AdvanceField, BackField**.
- *Control Flow Keywords*: **if, else, while, return**.
- *Reserved Variables* are used to represent key names: **$Enter, $Exit, $Cancl, $Help, $Cmd, $Tab**; to hold the error return value

from the most recent shell script: **$ERROR**; to hold the user's login name: **$LOGNAME**; to hold the user's home directory: **$HOME**; and to hold the user's default shell: **$SHELL**.

- *Field Coordinates* define the upper left and lower right relative location of a field. Menu and list fields can be defined with multiple columns.
- *Display Values/User Values* refer to the menu item values that are displayed versus the menu item values that are used by the program. These are accessed by the menu name suffixes **.dval** and **.uval**.

5

Programmatic Forms and Menus

This chapter describes how a developer can write a program in C that interacts with the user by means of forms and menus. Although the examples are in C, other programming languages such as FORTRAN, Pascal, BASIC, and COBOL are supported by means of altered versions of the calls described in this chapter. For information on using other languages than C with CTAM, see Chapter 7, "Using CTAM with COBOL, BASIC, and FORTRAN."

There are two ways a programmer can make use of forms whose definition resides in a *resource file*. The first way is to write the program so that the resource file is opened when the program runs. This allows application developers to provide single executable files whose interaction with the user can be customized in the field, since the resource file is an ASCII, editable file. For example, the text messages in the forms can be translated to a local language. In this type of application, the resource file is read and parsed by means of a special subroutine call at the beginning of the program.

The second way is to compile the form definition into the program. [See **rcc**(1).] Although the text in the form is not easily modified from session to session, this technique has the advantages of added program security and easier product management, since there is no external forms definition file to keep track of.

Using either method of building a form–based application program, the calls to display, access, and reset forms are the same. The resource file remains the same in both models of programming. The initialization calls differ slightly.

Compiling a DPL Program

A C program that accesses forms at runtime must be linked with the DPL and CTAM runtime libraries. This is usually done like this:

```
cc -o prog prog.c -ldpl -lctam
```

The Base Set of Forms Calls

This section outlines the set of calls that allow an application developer to build programs using forms. Chapter 4 describes the complete set of calls available.

Example 1

Below is a listing of a file **MyForms.rf**:

```
form MyFormName "Pick an Item" (2,3)
      field MyFieldName menu (2,2)
            "Menu Item #1",
            "Menu Item #2",
            "Menu Item #3";
```

Below is a listing of a file **prog.c**:

```
#include <dpl.h>

main()
{
      char FieldValue[80];
      form_t *MyFormName;

/* For readability, this example ignores error returns. */

      InitForms(0);
      WindowInit(0);
      OpenFormFile( "MyForms.rf" );
      GetFormPtr( "MyFormName", &MyFormName );

      PopupForm( MyFormName, 0 );
      GetFieldValue( MyFormName, "MyFieldName", FieldValue );
      NoteForm( "The value entered was $1", FieldValue, 0 );
```

```
        WindowExit(0);
    }
```

In this example, the programmer has written a short DPL form definition that he wants accessed when his program runs. The call to **WindowInit** prepares the screen for the forms session. The **InitForms** call sets up the forms subsystem. **OpenFormFile** opens the file containing the form definition. **GetFormPtr** returns a structure pointer enabling the use of subsequent forms and menuing calls. **PopupForm** displays the form, the name of which is provided as a parameter, then receives the user's input and resets the screen, returning control to the program when the **$Enter** key is pressed.

The **GetFieldValue** call returns the string that holds the value selected by the user. In this example, the returned value could be the string "Item #1", "Item #2", or "Item #3", depending on which item the user selected.

The **NoteForm** call pops up a form that displays a message and waits for the user to press a key.

Example 2

The following example uses the same form definition file, **forms.rf**, but uses the file in a compiled form; the resource compiler **rcc**(1) is used to link the form into the executable program. In addition, the example demonstrates the proper use of error returns.

Below is a listing of **MyFile.rf**:

```
form MyFormName "Pick an Item" (2,3)
    field MyFieldName menu (2,2)
        "Menu Item #1",
        "Menu Item #2",
        "Menu Item #3";
```

Below is a listing of a file **prog.c**:

```
#include <dpl.h>

main()
{
    char FieldValue[80];
    int err;
    form_t *MyFormName;

    InitForms(0);
    WindowInit(0);
    Init_MyFile();

    if (( err = GetFormPtr( "MyFormName", &MyFormName ) != 0){
        ErrorForm( "Trouble in GetFormPtr", 0 );
        WindowExit(1);
    }

    if (( err = PopupForm( MyFormName, 0 ) ) != 0 ){
        if ( err != ERR_CANCELED )
            ErrorForm( "Error in PopupForm", 0 );
        WindowExit(1);
    }

    if (( err = GetFieldValue( MyFormName, "MyFieldName",
                            FieldValue ) ) != 0 ) {
        ErrorForm( "Trouble in GetFieldValue", 0 );
        WindowExit(1);
    }

    NoteForm( "The value entered was $1", FieldValue, 0 );

    WindowExit(0);
}
```

Whenever a form is compiled into an application, the call to **Init_MyFile** must be made. The routine is created by the forms compiler, and the name is derived from the resource source file name. For every resource file used, an initialization call is required, where the name of that call is constructed with the name **Init_** and the main body of the file name (minus the **.rf** suffix).

Using the Forms Compiler

The **rcc** program converts a resource file to a standard CTIX object file, for linking into a program or for standalone use. If the **rcc** program is only given the name of a **.rf** file, it attempts to produce an executable **a.out**; otherwise, it produces an object file that can be linked with other object files. Here is an example of the command sequence used to build and run this program:

```
rcc prog.c MyFile.rf
./a.out
```

All of the same options that are legal for **cc** are legal on **rcc**.

Special Features

Since the form definition is compiled into the program using **rcc**, there are some additional action routine capabilities that are available. Recall that an *action routine* is a C syntax subroutine call like **doexec, SetPrompt,** or **dohelp**. The user can add his own action routines by writing them in C and calling them from his form definition. These are called *event traps*. For example:

```
form MyForm "Enter Your Name" (1,1)
     field MyName edit onselect { MySub( $MyForm.MyName ); }
          "";
```

In the above example, the user writes the C code for **MySub**, and **MySub** gets called after the edit field value is entered. The forms system treats these calls to C as if they were function calls that return a pointer. Thus, when the C code is written, it should be written to return either a legitimate pointer or a zero:

```
char *MySub( Param )
char *Param;
{
      .
      .
      .
      return( "Value" );
      .
      .
      .
      return( 0 );
}
```

When in doubt, make all **return** routines, into **return(0)**.

The forms system only passes parameters as strings. It allows up to 32 parameters.

Restrictions

When using **rcc**, no form names may be C program keywords. C programming keywords are: **auto**, **break**, **case**, **char**, **continue**, **default**, **do**, **double**, **else**, **entry**, **extern**, **float**, **for**, **goto**, **if**, **int**, **long**, **register**, **return**, **short**, **sizeof**, **static**, **struct**, **switch**, **typedef**, **union**, **unsigned**, and **while**.

6

The CTAM Internationalization Kit

This chapter is arranged as a tutorial for the software developer who wants to use the CTAM Internationalization Kit to develop software that works using a variety of natural languages. The CTAM Internationalization Kit enables the programmer to build applications with text messages, forms, and menus residing in files that are separate from the executable files that make up the final product. These messages are accessed at runtime by subroutines described in this chapter.

Note that the subroutines described in this chapter are modeled after the proposed X/OPEN Nationalization Standard. A subset of the X/OPEN capability is included.

Internationalization Subroutines

The following subroutines are contained in the library **/usr/lib/libxnls.a**:

nl_init

```
int nl_init( lang )
char *lang;
```

The **nl_init** routine is called before all other nationalization subroutines to establish the language of operation. If it is not called, subsequent subroutines operate in American English. It may only be called once.

The parameter **lang** points to a character string which defines the language for subsequent operations. For example:

```
nl_init( "french" );
```

By convention, the environment variable **$LANG** contains the name of the language of operation.

If the **lang** parameter is null, **nl_init** does the equivalent of this:

```
nl_init( getenv( "LANG" ) );
```

If successful, **nl_init** returns zero; minus one if it fails. It fails if **lang** does not contain the name of a valid language. To verify that the language is valid, **nl_init** attempts to open the directory **/usr/lib/lang/$LANG**, or, if the optional environment variable **$NLSPATH** is set, the directory **$NLSPATH/$LANG**.

nl_langinfo

```
#include <langinfo.h>

unsigned char *nl_langinfo( item )
int item;
```

The **nl_langinfo** routine returns a pointer to a null-terminated string containing information relevent to the language of operation. The values of **item** are defined in **langinfo.h** described in a later section of this chapter. For example:

```
nl_langinfo( ABDAY_1 );
```

would return a pointer to the string "Dom" if the language of operation was Portuguese, and "Sun" for English. **ABDAY_1**, refers to the abbreviation of the first day of the week.

The **nl_langinfo** call is typically used to obtain details about the current environment to be passed to subsequent nationalization calls.

Ctype

```
#include <nl_ctype.h>

int isalpha(c)
unsigned char c;
        .
        .
        .
```

These macros work the same as their counterparts in the standard CTIX release, except that they use an 8–bit character set. Each one returns non–zero for true, zero for false. Below is the complete list:

isalpha	**c** is a letter (upper–case or lower–case).
isupper	**c** is an upper–case letter.
islower	**c** is a lower–case letter.
isdigit	**c** is a digit (the characters [0–9]).
isxdigit	**c** is a hexadecimal digit (the characters [0–9], [A–F], or [a–f]).
isalnum	**c** is an alphanumeric (letter or digit).
isspace	**c** is a "white space" character (space, tab, return, newline, vertical tab, or formfeed).
ispunct	**c** is a punctuation character (any printing character except the space character, digits, and letters).
isprint	**c** is a printing character.
isgraph	**c** is a character with a visible representation (printing characters excluding "white space").
iscntrl	**c** is a control character (character codes less than 040, the 0177 character, codes in the range 0200–0237, 0377, or any other non–printing character).
isascii	**c** is an ASCII character, a code between 0 and 0177 (octal) inclusive.

conv

```
#include <nl_ctype.h>

unsigned char toupper(c)
unsigned char c;

unsigned char tolower(c)
unsigned char c;

unsigned char toascii(c)
unsigned char c;
```

These routines work like their counterparts in standard CTIX, except that they operate on an 8–bit character set. **Toupper** converts a character to upper case. **Tolower** converts a character to lower case. **Toascii** converts a character to 7–bit ASCII. For example, "A" umlaut is converted to "A".

ctime

```
#include <time.h>

char *nl_cxtime( clock, format )
long *clock;
char *format;

char *nl_ascxtime( tm, format )
struct tm *tm;
char *format;
```

Nl_cxtime extends the functionality of the standard CTIX **ctime** call by allowing date and time information to be output in a number of different ways, as defined by the **format**, and by providing month and weekday names in an appropriate form, as defined by the language of operation set up by the **nl_init** call.

The value of **clock** is obtained by means of the standard CTIX **time** call.

Format contains a string which is similar to the format strings given as the first argument to **printf**, where each field descriptor is preceded by % and is replaced in the output by its corresponding value. A single % is coded as %%. All other characters are copied to the output without change.

Below are the field descriptors:

n	insert a newline character
t	insert a tab character
m	month of year – 01 to 12
d	day of month – 01 to 31
Y	last 2 digits of year – 00 to 99
D	date as mm/dd/yy
H	hour – 00 to 23
M	minute – 00 to 59
S	second – 00 to 59
T	time as HH:MM:SS
j	Julian day – 001 to 366
w	day of week – 0 to 6 (Sunday is 0)
a	abbreviated weekday – Sun, Mon, Tue,...
h	abbreviated month – Jan, Feb, Mar,...
r	time in AM/PM notation

If **format** is null, the default time format for the language is used. Note that this is the same as the format returned by this call:

```
nl_langinfo( D_T_FMT )
```

Example:

```
        .
        .
        .
long clock;

nl_init( "italian" );
        .
        .
        .
clock = time();
printf( "%s", nl_cxtime( clock, "%a, %d %h 19%y %H:%M:%S" ) );
        .
        .
        .
```

The output value for Saturday, May 17, 1986 would be:

```
sab, 17 mar 1986 11:46:50
```

Nl_ascxtime converts a **tm** structure to a 26 character string, like **nl_cxtime**, also allowing the date string to be formatted. The **tm**

structure is obtained by means of the standard CTIX **localtime** or **gmtime** calls.

See the CTIX **ctime(3C)** routine for further information on handling of CTIX time and date.

OpenTextFile

```
int OpenTextFile( filename );
char *filename;
```

OpenTextFile opens a file containing text messages to be used by the program; these are the messages that are translated to a particular language. The messages are accessed by the program by means of the **GetText** function. **Filename** contains the name of the text file. If the file is successfully opened, **OpenTextFile** returns zero.

The message file contains a series of assignment statements of the form:

```
<variable name> = <text message>;
```

Where the variable name is any character string starting with dollar sign ($). The text message is surrounded by quotes. For example:

```
$Prompt1 = "Please enter your name";
```

For further examples, see the description of **GetText**.

Users of the DPL Forms and Menu package will note that the message file format is identical to the use of variables in the DPL language. If you want to use forms and menus along with the nationalization routines described in this chapter, you should define your messages in your *resource file* along with the form definitions, and use the **OpenFormFile** call instead of **OpenTextFile**.

OpenTextFile operates by opening the file, reading its contents into memory, building a symbol table, and then closing the file. By the time control is returned to the calling program, the file is closed; thus, the number of files allowed open by the program is unaffected.

GetText

```
unsigned char *GetText( VarName )
char *VarName;
```

GetText is used to retrieve a message from the message file following an **OpenTextFile** call. **VarName** contains the name of the variable associated with the message that is desired (**$Prompt1** in the previous example).

Below is the message file:

```
$msg = "This is my English message";
```

Example C program:

```
printf( "%s", GetText( "$msg" ) );
```

This is the resulting output:

```
This is my English message
```

nl_sprintf, nl_fprintf, nl_printf

```
int nl_printf( format[, arg ... ])
char *format;

int nl_sprintf( s, format[, arg ... ])
char *s, *format;

int nl_fprintf( stream, format[, arg ... ])
FILE *stream;
char *format;
```

These routines provide functionality similar to that of **printf, sprintf,** and **fprintf**, except that the conversion character % in **printf** is replaced by the sequence **%digit$**. (The following options should not be used: %e, %E, %f, %F, %g, %G.) **Digit** is a decimal digit **n** where conversions are applied to the **n**th argument in the argument list, rather than the next unused argument. This feature is used to allow text messages that are constructed at runtime to be language–independent with respect to word ordering.

Below is the message file:

```
$MyMessage = "Please insert the %1$s into the %2$s.";
$Disk = "disk";
$Drive = "disk drive";
```

Example:

```
char *format, *disk, *drive;
        .
        .
        .
format = GetText( "MyMessage" );
disk = GetText( "Disk" );
drive = GetText( "Drive" );
        .
        .
        .
nl_printf( format, disk, drive );
        .
        .
        .
```

The resulting message is:

```
Please insert the disk into the disk drive.
```

In a language other than English, however, a grammatically correct sentence may require that the words "disk drive" precede the words "insert the disk" in the sentence. If this is the case, the conversion specifications are reversed in the translated version of the message file. The program remains unchanged.

If it is not necessary to alter the order of conversion characters, the **digit** can be omitted.

nl_scanf, nl_sscanf, nl_fscanf

```
int nl_scanf( format[ ,pointer ...] )
char *format;

int nl_sscanf( s, format[, pointer ...] )
char *s, *format;

int nl_fscanf( stream, format[, pointer ...] )
FILE *stream;
char *format;
```

These routines provide functionality similar to that of **scanf, sscanf,** and **fscanf**, except that the conversion character % in **scanf** is replaced by the sequence **%digit$**. (The following options should not be used: %e, %E, %f, %F, %g, %G.) **Digit** is a decimal digit **n** where conversions are applied to the **n**th argument in the argument list, rather than the next unused argument. This feature is used to allow text messages that are constructed at runtime to be language-independent with respect to word ordering.

Nationalization Files

langinfo.h

This file is used along with **nl_langinfo** to retrieve information about the current language:

D_T_FMT	string for formatting the date and time. These are the "format descriptors" documented under nl_cxtime.
DAY_1	name of the first day of the week
.	
.	
.	
DAY_7	name of the last day of the week
ABDAY_1	abbreviated name of the first day of the week
.	
.	
.	
ABDAY_7	abbreviated name of the last day of the week
MON_1	name of the first month in the Gregorian calendar
.	
.	
.	
MON_12	name of the twelfth month
ABMON_1	abbreviated name of the first month
.	
.	
.	
ABMON_12	abbreviated name of the twelfth month
RADIXCHAR	radix character
THOUSEP	separator for thousands
YESSTR	affirmative response for yes/no questions

NOSTR negative response for yes/no questions

CRNCYSTR currency symbol, preceded by – if the symbol should appear before the value, or + if the symbol should appear after the value.

Language Configuration Database

The language configuration database is used by the nationalization libraries to obtain information about the current language. It resides in the directory **/usr/lib/lang**. You need to create this file for any language that your system does not yet support. You can modify this file to further customize for the needs of your installation.

The file makes use of keywords followed by equals signs to define the requirements of the language. The keywords are the same as those described for **langinfo.h**. Below is the example language database file **/usr/lib/lang/english_usa**:

```
D_T_FMT= "%a %h %d %T 19%Y";

DAY_1= "Sunday";
DAY_2= "Monday";
DAY_3= "Tuesday";
DAY_4= "Wednesday";
DAY_5= "Thursday";
DAY_6= "Friday";
DAY_7= "Saturday";

ABDAY_1= "Sun";
ABDAY_2= "Mon";
ABDAY_3= "Tue";
ABDAY_4= "Wed";
ABDAY_5= "Thu";
ABDAY_6= "Fri";
ABDAY_7= "Sat";

MON_1= "January";
MON_2= "February";
MON_3= "March";
MON_4= "April";
MON_5= "May";
MON_6= "June";
MON_7= "July";
MON_8= "August";
MON_9= "September";
MON_10= "October";
MON_11= "November";
```

```
MON_12= "December";

ABMON_1= "Jan";
ABMON_2= "Feb";
ABMON_3= "Mar";
ABMON_4= "Apr";
ABMON_5= "May";
ABMON_6= "Jun";
ABMON_7= "Jul";
ABMON_8= "Aug";
ABMON_9= "Sep";
ABMON_10= "Oct";
ABMON_11= "Nov";
ABMON_12= "Dec";

RADIXCHAR= ".";

THOUSEP= ",";

YESSTR= "yes";
NOSTR= "no";

CRNCYSTR= "+$";
```

The values used by the nationalization library follow the equals sign. Leading blanks are ignored.

7

Using CTAM with COBOL, BASIC, and FORTRAN

This chapter briefly outlines what you must do to use the programming libraries in the CTAM release with programming languages other than C.

COBOL

It is necessary to use a "wrapper" library in order to use the CTAM libraries from LPI COBOL. This is because the CALL USING construct always passes pointers to the parameters in COBOL, but the CTAM libraries often expect simple values instead of pointers. In addition, the use of strings in COBOL is inconsistent with the usage in C.

Linking CTAM with COBOL has been done, and a prototype wrapper library has been produced for that purpose. It is recommended that you contact your technical support representative to obtain the code for that wrapper for use as a template in building your own wrapper library.

Specifically, the wrapper performs these functions:

- The wrapper library addresses the fact that COBOL maps all subroutine names to upper case when it generates calls to access them. For example:

  ```
  CALL "OpenFormFile" USING erc, FileName.
  ```

  Gets turned into a call to OPENFORMFILE, in upper case.

- The wrapper library massages all strings to be null terminated, as expected by C.
- The wrapper returns the function return values from the CTAM library in a parameter. Since COBOL does not support function calls, this is the only way to obtain error codes. This example demonstrates a C function call to CTAM:

  ```
  erc = GetFormPtr( "FormName", &pForm );
  ```

  Here is the equivalent in COBOL:

  ```
  CALL "GETFORMPTR" USING erc, "FormName", pForm.
  ```

- The wrapper puts the function return value into the variable **erc**.

Using the wrapper library, compile your programs like this:

```
lplcobol myprog.cbl
lpild -o myprog libwrapper.a -ldpl -lctam
```

COBOL programs that use DISPLAY statements to output to the screen in addition to CTAM calls for the same function have trouble linking if their COBOL programs do cursor positioning as part of the DISPLAY statement. The reason is that the loader finds duplicate entry points when it links the CTAM libraries with the **termcap** libraries from CTIX. This can be avoided by modifying the **lpild** shell script.

Note that the typing of the parameters passed to the wrapper library is very strict. Form pointers and error return values should always be of type **PIC 9(8) COMPUTATIONAL**. All other parameters should be **PIC X** variables, where the size accommodates your needs. Note that strings returned by CTAM to the COBOL application contain null bytes at the end of the string. CTAM does not blank pad like COBOL.

BASIC

It is not necessary to use a "wrapper" library in order to use the CTAM libraries from BASIC. It is, however, necessary to build your own version of BASIC that incorporates the CTAM libraries. Rebuilding BASIC to call C subroutines is documented in the BASIC release notice. For a prototype **/usr/lib/basic/Basgen.config** file, contact your technical support representative.

Note that the calls to **WindowInit** and **InitForms** are allowed to be done only once by a program. The **WindowExit** call resets the screen to normal operation and terminates the application. Since BASIC is an interpreted environment, it is undesirable for an application to call **WindowExit**, since this does not allow the interpreter to do large amounts of necessary cleanup, such as flushing internal file buffers and scheduling spoolers, before exiting. As an alternative, the call **wprexec** resets the screen but leaves control in BASIC. This call should be used instead of **WindowExit**. Since **WindowInit** and **InitForms** should only be called once, it is difficult to use the BASIC RUN command to rerun a program without exiting BASIC and trying again. It is recommended that you write wrappers for **WindowInit** and **InitForms** that keep a flag to see if they have been called before, and if they have been, to bypass the **WindowInit** call:

```
int MyWindowInit()
{
        static flag = 0;

        if ( flag == 0 ) {
                flag == 1;
                return WindowInit();
        }
        return;
}

int MyInitForms()
{
        static flag = 0;

        if ( flag == 0 ) {
                flag = 1;
                return InitForms();
        }
        return;
}
```

Form pointers should be integer data types in BASIC. Strings work correctly with no massaging needed. If a string is going to be stored

by the CTAM libraries, you must first store dummy values into that string to allocate the space for CTAM. This is demonstrated in line 56 of the example below:

```
5 defint I, p
10 I = MyWindowInit
20 I = MyInitForms
25 x$ = "MyForm.rf"
30 I = OpenFormFile( ptr( x$ ) )
35 myform$ = "MyForm"
40 I = GetFormPtr( ptr( myform$ ), ptr( pForm ) )
50 I = PopupForm( pForm )
55 myfield$ = "MyField"
56 value$ = string$( 80, " " )
60 I = GetFieldValue( pForm, ptr( myfield$ ), ptr( value$ ) )
70 I = NoteForm( ptr( value$ ) )
80 I = GetWindowId( pForm, ptr( id ) )
90 I = wprexec( id )
100 end
```

FORTRAN

It is necessary to use a "wrapper" library in order to use the CTAM libraries from CTIX FORTRAN. This is because the subroutine calling construct always passes pointers to the parameters in FORTRAN, but the CTAM libraries often expect simple values instead of pointers. In addition, the use of characters in FORTRAN is inconsistent with the usage in C.

Linking CTAM with FORTRAN has been done, and a prototype wrapper library has been produced for that purpose. It is recommended that you contact your technical support representative to obtain code for that wrapper for use as a template in building your own wrapper library.

Specifically, the wrapper performs these functions:

- It gets around the restriction that FORTRAN maps all subroutine calls to lower case.
- It massages FORTRAN CHARACTER variables to conform to the needs of CTAM.

To use the wrapper library described here, it is necessary to have a FORTRAN compiler that supports the +cc compile time option, to allow easy calling from FORTRAN to C.

When designing your FORTRAN code to call CTAM, make all of the form pointers be INTEGER*4 variables. The functions return INTEGER*4 values. All strings should be FORTRAN CHARACTER variables. When you expect CTAM to return a string, you must first assign a value to that string to allocate the space for CTAM to overwrite:

```
Character*80 Value
      .
      .
      .
Value = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
call GetFieldValue( pForm, 'MyField', Value )
      .
      .
      .
```

A

Introduction to the CTAM Manual Pages

This appendix contains CTAM–related manual pages that are not included in the *CTIX Operating System Manual*. Following this introduction is a table of contents; the manual pages are presented alphabetically by section. Certain distinctions of purpose are made in the headings:

COMMANDS AND APPLICATION PROGRAMS:

(1) (DPL)	These are commands of general utility that are used with DPL resource files.
(1W) (CTAM)	These are commands used for invoking CTAM window management facilities.

SUBROUTINES:

(3W) (CTAM)	These functions constitute the CTAM library **libctam.a**, which is automatically loaded by the CTAM window manager [**ctwm**(1w)].
(3D) (DPL)	These functions constitute the DPL library **libdpl.a**, which is linked to resource files automatically by the forms interpreter [**dplrun**(1)].

FILE FORMATS:

(4W) (CTAM)	These entries describe the configuration files used with the CTAM package.

SPECIAL FILES:

(7W) (CTAM)	These entries describe special files that refer to specific hardware peripherals and CTIX system device drivers used by CTAM.

TABLE OF CONTENTS

1. Commands and Application Programs

ctwm . . . . CTAM Window Manager
dplrun . . . . interpret a resource file
rcc . . . . compile a resource file into an object file
wtty . . . . set window configuration for ctwm

3. Subroutines

wattron, wattroff, wattrset . . . . control window attributes
wflush . . . . write all pending window output to the screen
WGetArgs, WSetArgs . . . . window control operations
wgetc, keypad, wndelay, weof . . . . get the next keystroke
wgoto, wgetpos . . . . control cursor in a window
WindowCreate, PadWindowCreate . . . . create a new window
WindowInit, WindowExit . . . . begin and end windowing session
WindowLabel . . . . write to window special lines
WindowPrompt . . . . write to window special lines
WindowCmd . . . . write to window special lines
WindowSLK . . . . write to window special lines
wprexec, wpostwait . . . . suspend a CTAM process
wprintf . . . . output a formatted string in a window
wputc, wputs, wwrite . . . . output a character in a window
WSetSelect, WGetSelect . . . . select the current active window
AddItem . . . . programmatically construct a menu
ClearCmd . . . . clear menu type-ahead
ClearList . . . . deselect all items in a field
CloseForm . . . . clear a form from the screen
DisplayForm . . . . display a form
DupForm . . . . create a duplicate of a form
ErrorForm . . . . display a message in an error window
ExpandString . . . . expand embedded parameters in a string
GetCurrentField . . . . active field control
SetCurrentField . . . . active field control
AdvanceField . . . . active field control
BackField . . . . active field control
GetFieldValue . . . . retrieve a field's value
GetNextValue . . . . retrieve a field's value
GetFieldSelValue . . . . retrieve a field's value
GetFieldCmdValue . . . . retrieve a field's value
GetFormError, GetFormKey, GetWindowId . . . . get the state of a form
GetFormPtr . . . . get pointer to a form structure
GetLevel . . . . get capability level of forms session
GetString . . . . get a message string from a resource file
HideField, RevealField . . . . control appearance of a field
InitForm . . . . initialize programmatic menu construction

InitForms . . . . initialize the forms system
InputForm . . . . get input from programmatic menus
KeyForm . . . . feed keystrokes to a form
MessageOn . . . . leave a message in a window
NoteForm . . . . display a message in a window
OpenForm . . . . prepare a form to be displayed
OpenFormFile . . . . open resource file with forms definition
PopupForm . . . . display a form and get user input
RefreshField . . . . redisplay a field of a form
ResetForm . . . . prepare form for programmatic menu construction
SetEditDefault . . . . set the default edit field value
SetFormArgs . . . . set up arguments to a form
SetFormPos . . . . specify a form´s size and position
SetLevel . . . . set capability level of a forms session
SetListDefault . . . . set the default list item(s)
SetMenuDefault . . . . set the default menu item

4. File Formats

fonts . . . . CTAM font mapping files
kbmaps . . . . CTAM keyboard mapping files

7. Special Files

escape . . . . window escape codes

NAME

ctwm – CTAM Window Manager

SYNOPSIS

exec ctwm [-**r** visible rows] [-**c** visible columns] [-**x** start column] [-**y** start row] [-**h** height] [-**w** width] [-**e** switch key] [-**l** command file] [-**b**] [-**f**] [-**g**] [-**o**] [-**p**] [-**s**] [-**t**] [initial shell]

wexec [-**r** visible rows] [-**c** visible columns] [-**x** start column] [-**y** start row] [-**h** height] [-**w** width] [-**b**] [-**f**] [-**g**] [-**p**] [command]

wconfig [-**u** max. user windows] [-**s** max. super user windows]

DESCRIPTION

Ctwm is the CTAM window manager which enables multiple applications to run simultaneously on a terminal in multiple windows transparent to the application. With *ctwm* the output of several programs is coordinated for display on the user's terminal such that each application is confined to a particular rectangular region or "window" on the screen. Each window functions as an entire virtual display screen distinct from the other windows. Output sent to the screen by an application program is clipped by the window manager to fit in its window's *viewport*. The viewport size is defined by the number of rows and columns visible to the user between the window borders when the window is un-obscured by other windows. The size and placement of windows on the screen is arbitrary and completely under user control.

Application programs are often written to take advantage of an entire screen. CTAM supports full screen *pads*, where a pad is the screen area into which the viewport allows the user to see. Commands are available to scroll the pad, or change the viewport size to afford a full view of the contents of the pad. Full screen pads are stored by CTAM for every screen. As a result, programs that are written to use a whole screen work correctly unchanged under the windowing system. To determine individual terminal characteristics, the CTAM windowing system uses the *terminfo*(4) database as well as the appropriate file under the directories **/usr/lib/ctam/fonts** and **/usr/lib/ctam/kbmaps**. [See *fonts*(4W) and *kbmaps*(4W).]

The window manager supports features to manipulate windows. By pressing Control-Z (Code-Z on a Convergent PT or GT terminal), the user enters a mode in which all of his or her commands are directed to the window manager. The user can change Control-Z to any other character by setting the *switch key* option. In most cases, it is necessary to enter Control-D (Finish on a Convergent PT or GT) to exit the window manager and return control to the application running in the topmost window. When in window manager mode, the user's function keys are labeled to support the following features:

CREATE Create a window. When this function key is selected, the windowing system creates a new window with a user–defined window size. The shell defined by *initial shell* is spawned into that window.

SWITCH Switch topmost window. When this function key is selected, the arrow keys can be used to select the window that is to become the new topmost window. The topmost window is the window that receives input from the keyboard. The user presses the Down key to go forward through existing windows, Up to go backward. After selecting the new topmost window, the user enters Control-D, or Return, to return control to the application program running in that window. The user can also specify a particular window by either entering the window number (i.e. 0-9 where 0 is window #10) or the first letter of the window label. Control is automatically returned to the application program running in that window. If two or more windows have the same first letter, the window with the lowest window number is activated.

MOVE Moves a window. By selecting this function key, the user enters a mode where the arrow keys on the terminal are used to move the current topmost window. The arrow keys can be preceded by a number to move the window more than one slot at a time. When the user has placed the window in the desired location, entering Control-D or Return returns control to the application program running in that window. To move the window more than one space at a time, a number followed by an arrow key can be used.

GROW Grows a window. By selecting this function key, the user enters a mode where the arrow keys on the terminal are used to make the topmost window grow. Each arrow key grows the corresponding window border in that direction. Arrow keys can be preceded by a number to grow the window more than one slot at a time. When the user has grown the window to the desired size, pressing Control-D or Return returns control to the application program running in that window.

SHRINK Shrinks a window. By selecting this function key, the user enters a mode where the arrow keys on the terminal are used to make the topmost window shrink in size. Each arrow key shrinks the corresponding window border in that direction. The arrow key can be preceded with a number to shrink the window more than one slot at a time. When the user has shrunk the window to the desired size, pressing Control-D or Return returns control to the application program running in that window.

SCROLL Scrolls the pad. By selecting this function key, the user enters a mode where the arrow keys on the terminal are used

to scroll the display in the viewport. The user enters Control-D, or Return, to return control to the application program running in that window. To scroll the pad more than one space at a time, a number followed by an arrow key can be used.

MAX/PRE Size the window to the maximum or previous size. By selecting this function key, the user enters a mode where the window size is changed to the maximum or the previous size. The user enters Control-D, or Return, to return control to the application program running in that window.

MENU Displays a menu of all the existing window labels. The user selects the desired window and presses Return to return control to the application program running in that window.

TOPWIN Switch to top controlling window. This function replaces the CREATE function if the **-t** option is present. This is useful in the case where the user is only allowed to start new activities through a particular controlling window. This feature works well with the **-s** option (see below) in hiding the operating system from the user.

PASTE Paste characters into a window that have been CUT from another window. Any characters that are visible in a window can be marked and then pasted into another window. The window manager strips any attributes from the characters, such as inverse video or underline. When they are pasted into a window, they appear to the program running in that window as if the operator had typed those characters. The cut and paste procedure works like this: Press Control-Z to invoke the window manager. Select the window that you want to cut from. Press the Copy key (labelled F11 on a Convergent TO-235 or TO-250, Copy on a Convergent PT or GT). Next, move the cursor (using the arrow keys) to the start of the area that is to be cut. Press the Mark key (labelled Select on a TO-235/250, Mark on a PT/GT). Cursor to the end of the area that is to be cut. When you have marked the area to be cut, press Enter (labelled Do on a TO-235/250, Go on a PT/GT). Next, select the window to which you want to paste. Note that after you have CUT something, a function key label PASTE is displayed. Press this key to paste the cut data into the window you specified.

To refresh the screen's current contents, press Control-L (or Code-L). Control-C (or Code-C) pops up a menu of things to do on the screen.

To enter a shell command, type ! followed by the command and press Return. The command is echoed on the command line and the command is executed in a new window. After the command is finished, the user is prompted to acknowledge this. Control is returned to the next active window. This feature is disabled if the **-s** option is present.

Startup Options

The *initial shell* is the name of the shell that the user would like started up in all new windows created with the "CREATE" key. *Vrows* and *vcols* tell the windowing system how large to make new windows. *Start column* and *start row* describe the initial column and row position of the upper left corner of the first window. The upper left corner of the screen is at position (0,0).

If no sizing information is provided, *ctwm* defaults the window size to 22 rows by 78 columns, not counting the border characters. If *vrows* and/or *vcols* are set, but not the initial positions, when the windowing system creates a new window, it looks for a free area on the screen.

The **-b** option, if present, makes all windows borderless windows.

The **-f** option, if present, makes all windows fixed-size windows.

The **-o** option, if present, disallows the user from creating windows.

The **-s** option, if present, disallows the user from entering a shell command on the command line.

The **-t** option, if present, makes the resulting window from the corresponding shell command the top controlling window.

The **-p** option makes the pad and the viewport the same size.

The **-g** option prevents the window from scrolling to track the cursor.

If an application requires that the *pad* be a size other than that of the physical screen, the *height* and *width* options can be used to set the default. The *wtty*(1W) command can be used to change this from the shell. Programmatically, this can be changed by means of the *WSetArgs* [see *WGetArgs*(3W)] CTAM call.

The user can specify a list of commands to be executed in different windows when the windowing system first starts up by specifying the command list file name with the *command file* option. The format of the command list is:

```
[-r #] [-c #] [-h #] [-w #] [-x #] [-y #]
 [-p] [-b] [-f] [-t] shell command [options]
```

All the options correspond to the *ctwm* options. For example,

```
-r 4 -c 10 date; pwd; exec $SHELL
$SHELL
```

creates a 4 by 10 window executing a shell and a full size window executing another shell.

The windowing system supports a user-configurable background character to occupy the parts of the screen that do not contain a window. That character is defined by the optional environment variables **CTWM_BG** and **CTWM_ATTR**. For example, the following shell commands set the character "." as the background character, and run the window manager, starting the C-shell in a 10 by 20 window:

```
$ CTWM_BG=. ; export CTWM_BG
$ exec ctwm -h 10 -w 20 csh
```

If the **CTWM_BG** environment variable is not set, the background character defaults to blank. The environment variable **CTWM_ATTR** controls the attribute with which the background is displayed. The attribute is defined by giving the parameters to the **SGR** escape sequence [see *escape*(7W)] to be used with **CTAM_BG**. For instance, setting **CTWM_ATTR** equal to "7" would cause the background to be displayed in reverse video.

The windowing system also supports a user-configurable forms and menu file. The default file is **/usr/lib/ctam/english_usa/ctwm.rf**. This file contains all of the English text for the messages displayed by the window manager. The user can specify a customized file by setting the environment variable **CTWM_FORM**.

It is required that you start *ctwm* by using the shell's *exec* function (see example above). When the window manager is run, the user's **TERM** variable must be correctly set to the terminal on which the windowing system will run. Note that when the *initial shell* is spawned by the windowing system, the **TERM** variable is changed to reflect the requirements of *ctwm*. For example, the **TERM** variable *pt* is changed to *ptctam*. This must remain set in this way for correct operation of the windowing system. In addition, *ctwm* sets the **TERMCAP** variable to **CTWMtermcap**.

Window Signal

If an application running under the window manager wants to know when any of its windows is selected or re-sized by the user via the window manager, it should include a signal catching routine for **SIGWIND** (signal number 20). This signal is sent whenever a window becomes active (selected) or is re-sized by the window manager. If a program has multiple windows open it should call *WGetSelect* to see which, if any, of its windows is the active window. If the program has windows that can be re-sized by the user (windows without the **FIXEDSIZE** flag set in their window status structure), the program should call *WGetArgs* on each window to see what actually happened.

SEE ALSO

wtty(1W), terminfo(4), fonts(4W), kbmaps(4W).

WARNINGS

Ctwm is designed to be run from the host machine. Applications to be run over a network are supported in windows.

NAME

dplrun – interpret a resource file

SYNOPSIS

dplrun [**-s** StartingForm] file.rf [...file2.rf...] [**-c** opts...]

DESCRIPTION

Dplrun interprets a **resource file**, executing the menu and form information in the file. (For information on how to create a resource file, see Chapter 4 of this manual.) A resource file has the suffix **.rf**. The **-s** option provides the interpreter with the name of the starting form to display. If not used, the interpreter looks for the form named **mainform**.

The **-c** option allows the user to set the values of $1, $2, $3, etc., to be used by the starting form.

EXAMPLES

```
dplrun -s StartingForm MyFile.rf

dplrun MyFile.rf -c Hello World
```

NAME

rcc – compile a form

SYNOPSIS

rcc options

DESCRIPTION

Rcc compiles a **resource file** into a CTIX object file suitable for linking into "C" programs for execution. *Rcc* accepts the same set of options and file types as the *cc* command in addition to files ending with **.rf**. The libraries **libdpl.a** and **libctam.a** are automatically searched.

EXAMPLES

```
rcc -o myprog myforms.rf mycode.c

rcc -c myforms.rf
cc -o myprog mycode.c myforms.o -ldpl -lctam
```

NAME

wtty – set window configuration for ctwm

SYNOPSIS

wtty [**-height** height] [**-width** width] [**-vrows** vrows]
[**-vcols** vcols] [**-begx** begx] [**-begy** begy]
[**-border**] [**-fixedsize**] [**-padwin**] [**-track**]

DESCRIPTION

This command should be run from the shell inside a CTAM window. It is used to provide information about the window in which it is run or to alter the parameters for the window.

If given no parameters, *wtty* reports the beginning X and Y coordinates of the left hand corner of the window (*begx, begy*), the number of rows and columns in the viewport (*vrows, vcols*), the height and width of the pad (*height, width*), whether the window has borders, and whether the pad and the viewport are the same size. See the description of *ctwm*(1W) for a discussion on pads and viewports.

Height and *width* affect the size of the pad. *Vrows* and *vcols* affect the size of the viewport. *Begx* and *begy* affect the beginning coordinates: the upper left corner of the window. The **-border** flag causes the window to be displayed without borders surrounding it. The **-padwin** flag causes the pad and viewport to be "locked" together and always be equal. The **-fixedsize** flag prohibits the user from changing the window size with the window manager. The **-track** flag enables scrolling of the window to track the cursor.

SEE ALSO

ctwm(1W), WGetArgs(3W).

NAME

wattron, wattroff, wattrset – control window attributes

SYNOPSIS

```
#include <ctam.h>

int wattron( wid, onmask )
short wid;
short onmask;

int wattroff( wid, offmask )
short wid;
short onmask;

int wattrset( wid, onmask, offmask )
short wid;
short onmask;
short offmask;
```

DESCRIPTION

These calls provide a procedural interface to the CTSGR escape sequence. *Wattron* takes as arguments the window ID of the window to operate on and a bit-mask representing which attributes to be turned on. *Wattroff* takes the same arguments as *wattron* but the attributes are turned off instead of on. *Wattrset* takes as arguments the window ID to operate on and two bit-masks representing attributes to be turned on and off respectively. The following constants are defined in **ctam.h** and can be used to compute the mask: **A_UNDERLINE**, **A_REVERSE**, **A_BOLD**, **A_STRIKE**, and **A_DIM**.

SEE ALSO

escape(7W).

NAME

wflush – write all pending window output to the screen

SYNOPSIS

int wflush (wid)
short wid;

DESCRIPTION

Wflush causes the screen to be updated according to the current screen image in memory. Output to windows is normally buffered and flushed as a side effect of other CTAM calls such as *wgetc*(3W). Explicit calls to *wflush* can be used to cause a more immediate screen update. *Wflush* takes as an argument the window ID of the window to be flushed. When the program is not running under the CTAM Window Manager, a window ID value of -1 is treated as a special case and causes a terminal reset. A terminal reset involves clearing the screen and re-painting the windows that were on it. No application action is required to facilitate a reset.

DIAGNOSTICS

If the specified window ID does not correspond to a window currently open by the calling process, *wflush* returns EOF and sets *errno* accordingly.

SEE ALSO

wprintf(3W), wputc(3W), wgetc(3W).

NAME

WGetArgs, WSetArgs – window control operations

SYNOPSIS

```
#include <ctam.h>

int WGetArgs (wid, pwstat )
short wid;
struct wstat *pwstat;

int WSetArgs (wid, pwstat )
short wid;
struct wstat *pwstat;
```

DESCRIPTION

WGetArgs and *WSetArgs* allow certain aspects of a window to be determined and changed. These aspects include the window size and location on the screen. The parameters are a window ID as returned from a previous *WindowCreate*(3W) call and a pointer to a window status structure. A call to *WGetArgs* fills in the window status structure and a call to *WSetArgs* changes the specified window's settable parameters to the values passed in *pwstat*. The new parameters must be legal values with the same restrictions as those on the parameters passed in *WindowCreate*(3W).

The window status structure has the following form:

```
struct wstat {
        short    begy;           /* upper-left-corner row */
        short    begx;           /* upper-left-corner column */
        short    height;         /* number of logical rows */
        short    width;          /* number of logical columns */
        short    vrows;          /* number of visible rows */
        short    vcols;          /* number of visible columns */
        short    vcoff;          /* visible to logical column offset (RO) */
        short    vroff;          /* visible to logical row offset (RO) */
        short    crow;           /* current logical row of cursor (RO) */
        short    ccol;           /* current logical column of cursor (RO) */
        unsigned short uflags;   /* flags */
        };
```

The elements marked (RO) are read-only and attempts to change them with *WSetArgs* are ignored. The *uflags* element consists of the following flags ORed together:

NBORDER The window is to be displayed without a border.

FIXEDSIZE The window may not be re-sized by the window manager. The window may still be re-sized if a program executes a call to *WSetArgs*.

PADWIN The window is a "pad" window where the logical size does not necessarily match the visual size. The logical size is given by *height* and *width* and the visual size is given by *vrows* and *vcols*.

KBWIN The window is currently the active window if this flag is set. This flag is read-only and cannot be changed via *WSetArgs*.

EXAMPLES

```
/* Expected Results: The height and width of the window change.
*This is verified by WGetArgs.
*/
#include <ctam.h>
#include <stdio.h>

main ()
{
shortwindow;
struct wstatpwstat;
charbuffer[80];

(void)WindowInit(0);
if ((window = WindowCreate(1, 3, 15, 60, 0)) < 0)
{
fprintf(stderr, "WindowCreate failed\n");
exit();
}
wnl(window, 1);
sprintf(buffer, "WSetArgs Example");
WindowLabel(window, buffer);

if (WGetArgs(window, &pwstat) < 0)
{
fprintf(stderr, "WGetArgs failed\n");
WindowExit(1);
}
wprintf(window, "height = %d.\n", pwstat.height);
wprintf(window, "width = %d.\n", pwstat.width);
wprintf(window, "[Press return to continue.]");
(void)wgetc(window);

pwstat.height = 20; pwstat.width = 75;
if (WSetArgs(window, &pwstat) < 0)
{
fprintf(stderr, "WSetArgs failed\n");
WindowExit(1);
}
if (WGetArgs(window, &pwstat) < 0)
{
fprintf(stderr, "WGetArgs failed\n");
WindowExit(1);
}
wprintf(window, "\nheight = %d.\n", pwstat.height);
wprintf(window, "width = %d.\n", pwstat.width);
wflush(window);

wprintf(window, "[Press return to continue.]");
(void)wgetc(window);
WindowDelete(window); WindowExit(0);
}
```

WARNINGS

Programs that need to keep track of their windows' size and location should follow calls to *WSetArgs* with calls to *WGetArgs* to obtain the window's true status.

DIAGNOSTICS

WGetArgs and *WSetArgs* will return -1 upon failure.

SEE ALSO

WindowCreate(3W), WindowDelete(3W).

NAME

wgetc, keypad, wndelay, weof – get the next keystroke from the keyboard

SYNOPSIS

```
#include <ctam.h>

int wgetc ( wid )
short wid;

int keypad ( wid, flag )
short wid;
short flag;

int wndelay( wid, flag )
short wid;
short flag;

int weof( wid )
short wid;
```

DESCRIPTION

Wgetc returns the next keystroke from the window (i.e. the keyboard). The keyboard functions in "raw" mode, meaning that each key struck on the keyboard is returned to the caller without normal input processing. CTAM applications are expected to provide their own facilities for allowing users to correct simple typing mistakes. The function will not normally return until a keystroke is available. This routine is the window equivalent of *getchar*(3S) and provides a terminal independent method of reading from the keyboard.

Keypad is used to control the value returned by *wgetc*. *Wid* is ignored and *flag* is effective for all windows. The argument *flag* specifies one of four different modes of operation. If *flag* is 0, 7-bit mode is set. In 7-bit mode function keys return sequences of 7-bit characters from successive calls to *wgetc*. If *flag* is 1, 8-bit mode is set. In 8-bit mode function keys are returned as a single 8-bit value. The use of keypad mode 1 is not recommended. If *flag* is 2, unmapped or raw mode is set. In raw mode all keys are passed through untouched by CTAM. If *flag* is 3, 16-bit mode is set. In 16-bit mode function keys return a single 16-bit value. Mode 3 is the recommended mode (but see *WARNINGS* below).

Wndelay may be used to modify the behavior of *wgetc* when there are no keystrokes available. Normally, if *wgetc* is called when there are no keystrokes in the keyboard buffer, the process will sleep until the call can be satisfied. However, if a call is made to *Wndelay* with *flag* set, calls to *wgetc* will return **EOF** immediately if there are no keystrokes available. No delay mode can be disabled by calling *Wndelay* with *flag* reset (set to zero).

Weof returns zero if there are more keystrokes waiting in the keyboard buffer and non-zero otherwise.

EXAMPLES

First example:

```
/* Description:    Multi-window test where one window has no delay
 *                   and the other window has delay.
 *
 * Expected Results:        Wgetc should automatically return with
 *                          an EOF when there is no input pending in the
 *                          no-delay window and should block in the other.
 */
#include <ctam.h>
#include <stdio.h>

main ()
{        int       c, window1, window2;

         (void)WindowInit(0);

         if ((window1 = WindowCreate(3, 1, 8, 75, 0)) < 0)
         { fprintf(stderr, "WindowCreate failed\n"); WindowExit(1); }
         wnl(window1, 1);

         if (wndelay(window1, 1) < 0)
         { fprintf(stderr, "wndelay failed\n"); WindowExit(1); }

         WindowLabel(window1, "wgetc and wndelay Example: No-Delay Window");

         if ((window2 = WindowCreate(13, 1, 8, 75, 0)) < 0)
         { fprintf(stderr, "WindowCreate failed\n"); WindowExit(1); }

         wnl(window2, 1);
         WindowLabel(window2, "wgetc and wndelay Example: Blocking Window");
         wflush(window2);
         wputs(window1, "Whatever you type should be echoed back.\n");
         wputs(window1, "You may end the test by typing RETURN.\n");
win1:
         WSetSelect(window1);
         while ((c = wgetc(window1)) == EOF)
         {        wputs(window1, "Awaiting input.\n");
                  wflush(window1); sleep(3);
         }
         if (c == 015)
         {        WindowDelete(window1); WindowDelete(window2);
                  WindowExit(0);
         }
         else if (c == '2') goto win2;
         else     wputc(window1, c);
         while ((c = wgetc(window1)) != 015)
                  wputc(window1, c);
         wputc(window1, '\n'); goto win1;
win2:
         WSetSelect(window2);
         while ((c = wgetc(window2)) == EOF)
         {        fprintf(stderr, "FAILED:  Blocking window is no-delay.\n");
                  fflush(stderr);
                  WindowDelete(window1); WindowDelete(window2);
                  WindowExit(0);
         }
```

```
        if (c == 015)
        {       WindowDelete(window1); WindowDelete(window2);
                WindowExit(0);
        }
        else if (c == '1') goto win1;
        else    wputc(window2, c);
        while ((c = wgetc(window2)) != 015)
                wputc(window2, c);
        wputc(window2, '\n'); goto win2;
}
```

Second example:

```
/* Description:    Demonstrates the use of kcodes.h with keypad.
*/

#include <ctam.h>
#include <stdio.h>
#define BIGKEYS
#include <kcodes.h>

main ()
{       int     c, wid;
          .
          .
          .
        WindowInit(0);
        if (( wid = WindowCreate(...) < 0 )
          .
          .
          .
        keypad (0, 3);
        wprintf("Press a function key.\n");
        c = wgetc(wid);
        if ( c == EXIT )    /* Exit comes from kcodes.h.   */
                WindowExit();
        switch (c) {
                case F1:
              .
              .
              .
                case F2:
              .
              .
              .
                case F3:
              .
              .
              .
```

FILES

/usr/include/kcodes.h - defines constants for returned keystrokes

DIAGNOSTICS

Wgetc may return the integer constant **EOF** upon receipt of a signal or other error. *Wndelay* and *keypad* return -1 on failure and 0 otherwise.

WARNINGS

When using keypad mode 3 it is necessary to define the pre-processor constant "**BIGKEYS**" prior to including **kcodes.h**.

The *keypad* routine should not be called from DPL forms applications. Note that the use of *Wndelay* or the modification of *ioctl* parameters VMIN and VTIME [see *termio*(7)] may cause escape sequences to be broken. The result is that *wgetc* returns characters as if *keypad*(*wid*, 2) were called, regardless of the *keypad* mode.

BUGS

The *wid* argument to *keypad* is ignored and the value of *flag* applies to all windows.

NAME

wgoto, wgetpos – control cursor in a window

SYNOPSIS

```
int wgoto( wid, row, column )
short wid;
short row, column;

int wgetpos( wid, prow, pcolumn )
short wid;
short *prow, *pcolumn;
```

DESCRIPTION

Wgoto can be used to move the cursor to a specific row and column within the window. It is passed two coordinates that are zero-based (i.e., row 0, column 0, are the coordinates of the upper left hand corner of the window).

The *wgetpos* routine can be used find the current location of the cursor. *Wgetpos* is passed a window ID and pointers to two memory locations that will be filled with the cursor's row and column. Unlike *wgoto, wgetpos* uses one-based coordinates so that when the cursor is in the upper left hand corner of the window it will set row and column to (1,1).

DIAGNOSTICS

Both *wgoto* and *wgetpos* return -1 on failure.

NAME

WindowCreate, PadWindowCreate – create a new window

SYNOPSIS

```
#include <ctam.h>

int WindowCreate ( row, col, height, width, flags )
short row, col, height, width;
unsigned short flags;

int PadWindowCreate ( row, col, height,
vrows, vcols, flags )
short row, col, height, width, vrows, vcols;
unsigned short flags;

int WindowDelete( wid )
short wid;
```

DESCRIPTION

WindowCreate creates a new window on the screen of the current terminal. The windowing system must have been previously initialized with a call to *WindowInit*(3W). The first two arguments, *row* and *col*, give the character locations at which to place the upper left hand corner of the window. Note that if the window has borders, then the first available position on the window will be at location *row* +1, *col* +1. The next two arguments give the height and width of the window. The final argument is a modifier to the window. *Flags* may be any of the following ORed together:

NBORDER The window is to be displayed without a border.

FIXEDSIZE The window may not be re-sized by the window manager. The window may still be re-sized if a program performs a *WSetArgs*(3W) call on it or executes *wtty*(1W).

PADWIN The window is a "pad" window where the logical size does not necessarily match the visual size. The logical size is given by *height* and *width* and the visual size is given by *vrows* and *vcols*.

NEWPOS The resulting window is to be positioned in a new area of the screen, obscuring other windows as little as possible.

NTRACK The screen does not scroll to follow the cursor.

PadWindowCreate also creates a new window. However, unlike a window created with *WindowCreate*, the window will have a fixed virtual size and a flexible physical size. The two additional parameters specify the initial physical size (the visible rows and columns). A window created with *PadWindowCreate* will be re-sizable with the window manager.

Creating a window implicitly selects it as the topmost window.

WindowDelete removes the specified window from the screen possibly causing previously overlapped windows to become visible. The deleted window may no longer be written to with *wputc*.

EXAMPLES

```
/* Expected Result:   Borderless window displaying prompt.
 *
 */
#include <ctam.h>
#include <stdio.h>

main ()
{
        shortwindow;

        (void)WindowInit(0);

        if ((window = WindowCreate(3, 1, 10, 60,
        NBORDER|FIXEDSIZE)) < 0)
        {
        fprintf(stderr, "WindowCreate failed\n");
        WindowExit(1);
        }

        wprintf(window, "[Press return to continue.]");
        wflush(window);
        (void)wgetc(window);
        WindowDelete(window);
        WindowExit(0);
}
```

SEE ALSO

wputc(3W), wprintf(3W), WGetArgs(3W), wgetc(3W), escape(7W).

DIAGNOSTICS

WindowCreate and *PadWindowCreate* both return a new window ID upon successful completion and -1 upon failure. *WindowDelete* returns 0 if successful and -1 otherwise.

WARNINGS

If the requested window location and size would cause the window to extend off the screen, the window may be created with a different number of rows or columns than requested. *WGetArgs*(3W) should be used to determine the actual size and location of the window.

NAME

WindowInit, WindowExit - begin and end CTAM windowing session

SYNOPSIS

```
#include <ctam.h>
void WindowInit(0)

void WindowExit(ec)
short ec;

int iswind()
```

DESCRIPTION

WindowInit initializes a CTAM windowing session on *stdin* and *stdout*. *Stdin* is presumed to be a terminal device. CTAM looks for an environment variable **TERM** which specifies the *terminfo*(4) name for a terminal using the conventions of the windowing system. For example, the **TERM** variable *vt100ctam* is used to support VT-100 terminals under the windowing system.

The terminal's screen is cleared in preparation for windowing activities.

WindowExit terminates a window session and the current process. The parameter *ec* is passed to *exit* (2) and becomes the process exit value.

The *iswind* call may be used after a call to *WindowInit* to determine whether or not the application is running within windows [i.e., the CTAM Window Manager *ctwm*(1W) is already running]. If the program is executing with windows, *iswind* returns non-zero; zero otherwise. Some programs may want to alter their behavior when running under the window manager by catching the window signal **SIGWIND**.

FILES

/usr/lib/terminfo/*/* - Terminal descriptions
/usr/lib/ctam/kbmaps/*.kb - Keyboard descriptions
/usr/lib/ctam/fonts/*.ft - font descriptions

SEE ALSO

terminfo(4), font(4W), kbmap(4W).

DIAGNOSTICS

If CTAM is unable to determine the terminal type, or obtain a good terminfo description for the terminal, a message is printed and the program is terminated.

WARNINGS

Between calls to *WindowInit* and *WindowExit*, output to the terminal should not occur through any other means than calls to CTAM. When *WindowInit* is called, a check is made to see whether or not some signals are being caught. CTAM provides default signal handlers for several signals if they are not already being caught or ignored.

NAME

WindowLabel, WindowPrompt, WindowCmd, WindowSLK - write to window special lines

SYNOPSIS

```
#include <ctam.h>

int WindowLabel ( wid, label )
short wid;
char *label;

int WindowPrompt ( wid, ptext )
short wid;
char *ptext;

int WindowCmd ( wid, ptext )
short wid;
char *ptext;

int WindowSLK ( wid, kn, dummy, shortlabel, slkstring )
short wid;
short kn;
char *shortlabel;
char *slkstring;
```

DESCRIPTION

WindowLabel writes the string pointed to by *label* to the window specified by *wid*. The string is displayed along the top of the window. *WindowPrompt* writes the string pointed to by *ptext* to the prompt line of the window specified by *wid*. *WindowCmd* writes the string pointed to by *ptext* to the command line of the window specified by *wid*. *WindowSLK* writes the string pointed to by *shortlabel* to the window's function key label line, key number *kn*.

SEE ALSO

escape(7W).

NAME

wprexec, wpostwait – suspend a CTAM process

SYNOPSIS

```
int wprexec( wid )
short wid;

int wpostwait( wid )
short wid;
```

DESCRIPTION

These calls can be used when it is desired to suspend a CTAM application's access to the terminal to allow another process to take over. The *wprexec* routine is called by the parent *before* it forks to allow CTAM to "remember" the terminal's state and reset the terminal to normal shell modes. It takes a single argument, *wid,* the window number that the child process will use for its output.

After the child process has completed, a call to *wpostwait* resets the window specified by *wid* to its previous state. When running under the CTAM Window Manager *ctwm*(1W), the contents of the window will be lost. In such cases it is the responsibility of the application to then restore the window's contents. An application not running under the window manager should call *wflush*(3W) with a window ID of -1 to cause the screen to be re-painted.

DIAGNOSTICS

Upon failure these calls return a value of -1 to the application. Reasons for failure include passing a window ID that does not correspond to a window currently open by the application.

NAME

wprintf – output a formatted string in a window

SYNOPSIS

```
#include <ctam.h>

int wprintf ( wid, fmt, args . . . )
short wid;
char *fmt;
```

DESCRIPTION

Wprintf writes a formatted string to a window. The first parameter must be a window ID returned by a previous call to *WindowCreate*. The format string has the same form as in *printf*(3S).

SEE ALSO

wputc(3W), wflush(3W), escape(7W).

NAME

wputc, wputs, wwrite – output a character to a window

SYNOPSIS

```
#include <ctam.h>

int wputc (wid, c )
short wid;
char c;

int wputs (wid, str )
short wid;
char *str;

int wwrite (wid, buff, len )
short wid;
char *str;
int len;

int wnl ( wid, flag )
short wid;
short flag;
```

DESCRIPTION

The calls *wputc*, *wputs*, and *wwrite* are used to control the contents of a window. All characters passed to CTAM by these calls are interpreted by CTAM to effect the display of text, erasure of text, cursor movement, and highlighting of text. [See *escape*(7W) for a complete list of recognized character sequences.] Calls to these routines are buffered and do not actually take effect until the buffer is flushed either implicitly by calls to some other CTAM routines (such as *wgetc*(3W) or explicitly by a call to *wflush*(3W).

Wputc writes a single character to the specified window. The first parameter *wid* must be a window ID that was returned by a previous call to *WindowCreate*(3W).

Wputs is identical to *wputc* except that it writes a null-terminated string to the specified window.

Wwrite writes *len* bytes of the buffer pointed to by *buff* to the window specified by *wid*. This call can be used to write a buffer that may contain nulls to a window. The nulls are ignored by CTAM.

Wnl can be used to control the effect of the new-line character. If *flag* is non-zero, a new-line character (octal 012) is displayed as a linefeed followed by a carriage return. If *flag* is zero, a new-line character is simply displayed as a linefeed.

WARNINGS

The default for newline mapping is off. That is, a newline character output to a window generally only moves the cursor straight down one line. Many programs will want to follow a call to *WindowCreate* with a call to

wnl, setting *flag* to one. For instance, the following piece of code will probably not have the expected results:

```
    .
    .
    .
wid1 = WindowCreate(1,1,10,60,0);
wprintf(wid1, "Hello, world\n");
wprintf(wid1, "Goodbye, world\n");
    .
    .
    .
```

The output produced by these calls is the two lines of text, the first starting in column zero, and the second starting in column 13. The programmer should either insert a call to *wnl* before the print statements or should include a '\r' after '\n'.

SEE ALSO

wprintf(3W), wflush(3W), WindowCreate(3W).

NAME

WSetSelect, WGetSelect – select the current active window

SYNOPSIS

int WSetSelect (wid)
short wid;

int WGetSelect( )

DESCRIPTION

WSetSelect causes the window whose window ID is *wid* to become the current active window. The active window is brought to the front of the screen. For a discussion on determining the current active window, see *ctwm*(1W).

WGetSelect returns the window ID of the currently selected window.

EXAMPLES

```
/*  Expected Results:       Create several windows.  Interactively select
 *                          the new window.
 */
#include <ctam.h>
#include <stdio.h>

main ()
{           int         i, c;
            int         again = 1;
            short       row, col, window[5];

            (void)WindowInit(0);

/*  Create five windows and verify after each call.  */

            for (i = 0, row = 1, col = 3; i < 5; ++i, row += 3, col +=3)
            {
                        if ((window[i] = WindowCreate(row, col, 10, 50, 0)) < 0)
                        {
                                    fprintf(stderr, "WindowCreate failed\n");
                                    WindowExit(1);
                        }
                        wnl(window[i], 1);
                        WindowLabel(window[i], "WSetSelect Example");
                        wflush(window[i]);
            }
            i -= 1;
            wprintf(window[i], "This is the currently selected window.\n");
            wprintf(window[i], "It should be completely exposed.\n");
            wprintf(window[i], "Type a number from 1 to 5, or Return to exit.\n");

/*  Change window selection.  */

            while (again)
            {
            if ((c = wgetc(window[i])) < '1' || c > '5')
                        if (c == 015)
                        {
                                    for (i = 0; i < 5; i++)
                                    {
                                                WindowDelete(window[i]);
```

```
                                WindowExit(0);
                        }
                        again = 0;
                }
                else
                {
                        wputc(window[i], c);
                        wputc(window[i], '\n');
                        wprintf(window[i], "Try again, punk.\n");
                }
        else if ( c > '0' && c < '6' )
                {
                i = c - 49;          /* ASCII conversion     */
                WSetSelect(window[i]);
                if (WSetSelect(window[i]) < 0)
                        {
                        fprintf(stderr, "WSetSelect failed.\n");
                        WindowExit(1);
                        }
                wprintf(window[i], "Type 1 - 5, or Return to exit.\n");
                }
        }
(void)wgetc(window[i]);
WindowExit(1);
}
```

DIAGNOSTICS

Both routines return -1 upon failure. *WSetSelect* fails if the window ID to be selected is not currently associated with any window. *WGetSelect* fails if there are no active windows, or if the currently active window is not associated with the current process.

NAME

AddItem – programmatically construct a menu

SYNOPSIS

```
#include <dpl.h>
int AddItem( pForm, FieldName, NewValue, UserValue, flag )
form_t *pForm;
char *FieldName;
char *NewValue;
char *UserValue;
ushort flag;
```

DESCRIPTION

AddItem is used to add items to an existing field. *FieldName* is the name of the field in the form to which the field items will be added. *NewValue* is the field item value. *UserValue* is the optional value to return when the field item is selected. It is analogous to the value on the right hand side of the equals sign in a DPL program:

```
field menu
        "Item1" = "Value 1",
        "Item2" = "Value 2",
        .
        .
        .
```

If *flag* is set to the value ISELECTED, this field item is selected as the default.

This call, along with the *ResetForm*, *DisplayForm*, *InputForm*, and *CloseForm* calls, substitutes for the *PopupForm* call that is used in applications that do not build their own field items.

NAME

ClearCmd – clear menu type-ahead

SYNOPSIS

```
#include <dpl.h>
int ClearCmd( pForm, FieldName )
form_t *pform;
char *FieldName;
```

DESCRIPTION

ClearCmd is used by programs to control a field's command line. When the user types characters in a menu or list field, those characters are echoed on the command line. This call erases that line.

NAME

ClearList – deselect all items in a field

SYNOPSIS

```
#include <dpl.h>
int ClearList( pForm, FieldName )
form_t *pform;
char *FieldName;
```

DESCRIPTION

ClearList may be used by programs to de-select all items in a list.

NAME

CloseForm – clear a form from the screen

SYNOPSIS

```
#include <dpl.h>
int CloseForm( pForm )
form_t *pForm;
```

DESCRIPTION

CloseForm is used to clear the form from the screen after values have been selected by the user by means of the *InputForm* call. This call is frequently used in conjunction with the programmatic menu construction calls such as *AddItem*, *ResetForm*, *InitForm*, and *InputForm*.

OpenForm, *InputForm*, and *CloseForm* can also be used to substitute for a *PopupForm* call, where the program can check data values for correctness before clearing the form from the screen.

EXAMPLES

```
#include <dpl.h>

main()
{
        form_t *MyForm;
        char FieldValue[80];
        int err;
        int Finished;

        InitForms();
        Init_MyFile();
        WindowInit(0);
        GetFormPtr( "MyForm", &MyForm );

        Finished = 0;
        do {
          OpenForm( MyForm );
          InputForm( MyForm );
          GetFieldValue( MyForm, "MyField", FieldValue );
          if ( strcmp( FieldValue, "The Right Answer" ) != 0 )
                  NoteForm( "Wrong answer.  Try again.", 0 );
          else
                  Finished = 1;

        } while ( Finished != 1 );

        CloseForm( MyForm );
        WindowExit( 0 );
}
```

NAME

DisplayForm – display a form

SYNOPSIS

```
#include <dpl.h>
int DisplayForm( pForm )
form_t *pForm;
```

DESCRIPTION

DisplayForm is used in a mode where the programmer is using the advanced features of the forms system to substitute for the *PopupForm* call. *DisplayForm* updates a form by displaying new values that have been added to a field by means of the *KeyForm* call.

NAME

DupForm – create a duplicate of a form

SYNOPSIS

```
#include <dpl.h>

int DupForm( pForm1, &pForm2 )
form_t *pForm1;
form_t **pForm2;

int DestroyForm( pForm )
form_t *pForm;
```

DESCRIPTION

DupForm creates a new form that is a duplicate of another form. The new form will have all of the same characteristics as the original form in its reset state. This call allocates memory for the second form. It does not affect the screen.

DestroyForm should be used to reclaim the memory used by a form created with *DupForm*.

EXAMPLES

```
#include <dpl.h>

main()
{
        form_t *MyForm, *MyForm2;
        char FieldValue[80];
        int err;

        InitForms();
        WindowInit();
        OpenFormFile( "test501.rf" );

        GetFormPtr( "MyForm", &MyForm );

        DupForm( MyForm, &MyForm2 );

        ResetForm( MyForm );
        SetFormArgs( MyForm, "I am form #1" );
        InitForm( MyForm );
        InputForm( MyForm );

        ResetForm( MyForm2 );
        SetFormArgs( MyForm2, "I am form #2" );
        InitForm( MyForm2 );
        InputForm( MyForm2 );

        CloseForm( MyForm );
        CloseForm( MyForm2 );

        DestroyForm( MyForm2 );

        WindowExit();
}
```

NAME

ErrorForm – display a message in an error window

SYNOPSIS

int ErrorForm(format [, arg] ..., 0)
char *format;

DESCRIPTION

ErrorForm creates a "popup window," and displays an error message in it, waiting for the user to press a key before restoring the screen to its state prior to the message. The *format* value is a character string with an embedded $1, $2, $3, etc. into which the *arg* values are embedded. Like *PopupForm*, the error value could be the value ERR_CANCELED, if the Cancel key is used to exit the form.

EXAMPLES

```
        .
        .
        .
ErrorForm( "You had an error in this item: $1", ItemString, 0 );
        .
        .
        .
```

NAME

ExpandString – expand embedded parameters in a string

SYNOPSIS

```
unsigned char *ExpandString( fmt, [ arg, ... ] )
char *fmt;
char *arg;
```

DESCRIPTION

ExpandString may be used to expand embedded parameters in a string. The format string *fmt* may contain the embedded positional parameters $1, $2, $3, etc. for which the *arg* values are substituted. In addition, the format string may include embedded DPL variables. (For more information on embedded variables, see "Variables," in Chapter 4 of this manual.)

WARNINGS

ExpandString returns a pointer to a static buffer that is overwritten on subsequent calls.

EXAMPLES

```
char *s;
          .
          .
          .
s = ExpandString( "The $1 in $2", "rain", "Spain", 0 );
          .
          .
          .
fprintf(logfile,"Message: %s\n", s );
          .
          .
          .
```

NAME

GetCurrentField, SetCurrentField, AdvanceField, BackField – active field control

SYNOPSIS

```
#include <dpl.h>

char *GetCurrentField( pForm )
form_t *pForm;

int SetCurrentField( pForm, FieldName )
form_t *pForm;
char *FieldName;

int AdvanceField( pForm )
form_t *pForm;

int BackField( pForm )
form_t *pForm;
```

DESCRIPTION

These calls may be used to control which field in a form is the currently active field.

GetCurrentField returns a pointer to the name of the current field that is selected. *SetCurrentField* causes the given field to be selected. *AdvanceField* moves control to the next field; *BackField* to the previous field.

NAME

GetFieldValue, GetNextValue, GetFieldSelValue GetFieldCmdValue – retrieve a field's value

SYNOPSIS

```
#include <dpl.h>
int GetFieldValue(pForm, FieldName, Value)
form_t *pForm;
char *FieldName;
char Value[size];

int GetNextValue( pForm, FieldName, Value )
form_t *pForm;
char *FieldName;
char Value[size];

int GetFieldCmdValue( pForm, FieldName, Value )
form_t *pForm;
char *FieldName;
char Value[size];

int GetFieldSelValue( pForm, FieldName, Value )
form_t *pForm;
char *FieldName;
char Value[size];
```

DESCRIPTION

GetFieldValue returns the value entered by a user during a *PopupForm* call. *PForm* and *FieldName* identify the form and field. *Value* receives the value entered or selected by the user. The *size* is determined by the programmer with a maximum defined in **dpl.h** of **MAXSTRLEN**. The type of the value is always a character string, null terminated. *Value* is either the text value of the item selected, the text value of the first list item selected, the text value of the actual return value (if the equals sign was used on the item), the entered text for a *writein* menu item, or the value entered in an edit field.

To get the multiple values from a list, use *GetFieldValue* for the first call, and *GetNextValue* for subsequent calls.

GetNextValue is used like *GetFieldValue*, except that it is used to get some of the values entered in a **list** field. In normal use, *GetFieldValue* is used to get the first value, and *GetNextValue* to get the subsequent values, until the error return is ERR_NOSELECT.

For menus and lists, the **[writein]** feature is used to allow users to enter values that do not appear as items in the form. The *GetFieldValue* call returns the value typed, if one was entered by the user. Under some circumstances, it is required that the application know what menu item was selected even though the user entered a value. *GetFieldSelValue* serves this function, both when **[writein]** was used in the form and when it is not used. *GetFieldCmdValue* is used to return the value entered by the user.

EXAMPLES

```
form MyForm "Pick A File" (2,2)
        field FileName menu (2,2)
                'ls | sort';


#include <dpl.h>
main()
{       int err;
        form_t *MyForm;
        char FieldValue[80];

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        Init_MyFile();
        WindowInit(0);
        if (( err = PopupForm( MyForm, 0 ) ) != 0 ) {
                ErrorForm( "Forms call error ", 0 );
                WindowExit(1);
        }
        err = GetFieldValue(MyForm,"FileName",FieldValue);
        if ( err !=0 ){
                ErrorForm( "Can't get field ", 0 );
                WindowExit(1);
        }
          .
          .
          .
}
```

Using *GetNextValue* multiple items may be retrieved:

```
form mainform
        field list (2,2) "Item1", "Item2", "Item3", "Item4", "Item5";
        field text (8,2) "Choose one or more of these items";

#include <dpl.h>
main()
{
        form_t *mainform;
        char FieldValue[80];

        InitForms();
        WindowInit(0);
        OpenFormFile( "MyFile.rf" );
        GetFormPtr( "mainform", &mainform );

        PopupForm( mainform, 0 );
        GetFieldValue( mainform, "MyField", FieldValue );
        NoteForm( "Here's the first one: $1", FieldValue );
        while( GetNextValue( mainform, "MyField", FieldValue )
                                        != ERR_NOSELECT)
                NoteForm("Here's the next one: $1", FieldValue,0);

        WindowExit(0);
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

Chapter 4, "Introduction to DPL Programming," in this manual.

NAME

GetFormError, GetFormKey, GetWindowId – get the state of a form

SYNOPSIS

```
#include <dpl.h>

int GetFormError( pForm )
form_t *pForm;

int GetFormKey( pForm )
form_t *pForm;

int GetWindowId( pForm, pWid )
form_t *pForm;
short *pWid;
```

DESCRIPTION

GetFormError returns the last error from a form call.

GetFormKey returns the value of the key that was used to exit the last form. The value of this key can be found in **/usr/include/kcodes.h**.

GetWindowId returns the value of the CTAM window pointer that corresponds to the given form.

EXAMPLES

```
#include <dpl.h>

main()
{
        form_t *mainform;
        char FieldValue[80];
        int err;
        short ID;
        char c;

        InitForms();
        WindowInit(0);
        OpenFormFile( "test333.rf" );

        GetFormPtr( "mainform", &mainform ) ;
        OpenForm( mainform );
        DisplayForm( mainform );

        GetWindowId( mainform, &ID );
        while( ( c = wgetc( ID ) ) != '.' ) {
                KeyForm( mainform, c );
                DisplayForm( mainform );
        }
        GetFieldValue( mainform, "name", FieldValue );
        NoteForm( "Here's the field: $1", FieldValue, 0 );
        CloseForm( mainform );
        WindowExit(0);
}
```

NAME

GetFormPtr – get pointer to a form structure

SYNOPSIS

```
#include <dpl.h>
int GetFormPtr( FormName, pFormPtr )
char *FormName;
form_t **pFormPtr;
```

DESCRIPTION

GetFormPtr is used to obtain a form pointer for subsequent calls to the forms and menuing system. *PFormPtr* is a 32-bit pointer, and can be substituted with a *PIC 9(8) COMPUTATIONAL* variable in Cobol, an *INTEGER*4* in Fortran, a *longint* in Pascal, or an *integer* in Basic.

EXAMPLES

```
#include <dpl.h>
main()
{       int err;
        form_t *pForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        WindowInit(0);
        if (( err = OpenFormFile( "MyForms.rf" ) != 0 ) {
                ErrorForm( "Problem with the forms file ", 0 );
                WindowExit(1);
        }
        if (( err = GetFormPtr( "MyForm", &pForm ) ) != 0 ) {
                ErrorForm( "GetFormPtr error ", 0 );
                WindowExit(1);
        }
        if (( err = PopupForm( MyForm, 0 ) ) != 0 ) {
                ErrorForm( "Forms call error ", 0 );
                WindowExit(1);
        }
        WindowExit(0);
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**. Likely causes of failure include not having the requested form in the symbol table, in which case ERR_NOFORM is returned.

SEE ALSO

OpenFormFile(3D).

NAME

GetLevel – get capability level of forms session

SYNOPSIS

int GetLevel()

DESCRIPTION

GetLevel is used to obtain the **level** value that is used with the *level* item attribute in a resource file.

EXAMPLES

```
form mainform
        field menu
                "For Superusers"      [level=0],
                "For important users" [level=1],
                "For all users"       [level=7];
                  .
                  .
                  .

#include <dpl.h>

main()
{
        int level;
          .
          .
          .
        level = GetLevel();
          .
          .
          .
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

SetLevel(3D).

NAME

GetString – get a message string from a resource file

SYNOPSIS

```
#include <dpl.h>
unsigned char *GetString( VarName );
char *VarName;
```

DESCRIPTION

GetString is used to obtain a string constant from a *resource file*, presumably for use in a message to the user. Embedding string constants in a resource file and using OpenFormFile to access the file allows all user messages to be field upgradable, for things such as translation to a local language. *VarName* is the name of a variable in the resource file.

EXAMPLES

```
$GreetingMessage = "Guten Tag";
form mainform "The rest of the form file" (2,2)
            .
            .
            .

#include <dpl.h>
main()
{
        int err;
        form_t *pForm;

        InitForms();
        WindowInit(0);
        OpenFormFile( "MyForms.rf" );
        GetFormPtr( "MyForm", &pForm );
        NoteForm( GetString( "GreetingMessage" ), 0 );
            .
            .
            .
}
```

NAME

HideField, RevealField – control field's appearance

SYNOPSIS

```
#include <dpl.h>

int HideField( pForm, FieldName )
form_t *pForm;
char *FieldName;

int RevealField( pform, FieldName )
form_t *pForm;
char *FieldName;
```

DESCRIPTION

HideField may be used to completely remove a field from a form. A field that is hidden in this way will not appear when the form is displayed and cannot be made into the current field. This call is useful for situations where a field in an existing form is not needed.

In the example below, the form asks for the user to enter the time of day. If the application is run in a country that uses a 24 hour clock, the menu for 'AM' and 'PM' will not appear.

EXAMPLES

```
form TimeOfDay "Test 502" (2,2)
        field text (2,2) "Hour:   ";
        field hour edit (2,10)-(2,15) "";
        field text (4,2) "Minute: ";
        field minute edit (4,10)-(4,15) "";
        field AMPM menu (6,2) "AM", "PM";

#include <dpl.h>

main()
{
        form_t *MyForm, *MyForm2;
        char FieldValue[80];
        int err;
          .
          .
          .
        InitForms();
        WindowInit();
        OpenFormFile( "test502.rf" );

        GetFormPtr( "TimeOfDay", &MyForm );
        ResetForm( MyForm );
        if ( LanguageMode == GERMAN )
                HideField( MyForm, "AMPM" );
        InitForm( MyForm );
        InputForm( MyForm );
        CloseForm( MyForm );
        WindowExit();
}
```

NAME

InitForm – initialize programmatic menu construction

SYNOPSIS

```
#include <dpl.h>
int InitForm( pForm )
form_t *pForm;
```

DESCRIPTION

InitForm is used to prepare the form for programmatic menu construction, by means of the *AddItem* call and other calls. It creates the window for the form.

This call, along with the *ResetForm*, *DisplayForm*, *InputForm*, *AddItem*, and *CloseForm* calls, substitutes for the *PopupForm* call that is used in applications that build their own menu items.

NAME

InitForms – initialize the forms system

SYNOPSIS

int InitForms()

DESCRIPTION

InitForms prepares the forms system for subsequent forms and menuing calls. It should be called before *WindowInit* and should be called only once by a program. Returns an error code if called a second time.

If the form is compiled and linked into the application by means of the forms compiler *rcc*, it is necessary to perform further initialization. An initialization call for every resource file used is required. That call would be, for example, *Init_xyz*, where the resource file is named **xyz.rf**.

EXAMPLES

```
#include <dpl.h>

main()
{         int err;
          form_t *MyForm;

          if (( err = InitForms() ) != 0 ) {
                    printf( "Forms system error \n" );
                    exit( 1 );
          }
          Init_MyFile();   /*  assume the form is in MyFile.rf   */
          WindowInit(0);

          GetFormPtr( "MyForm", &MyForm );
          if (( err = PopupForm( MyForm, 0 ) ) != 0 ) {
                    ErrorForm( "Forms call error ", 0 );
                    WindowExit(1);
          }
          WindowExit(0);
}
```

FILES

/usr/lib/ctam/english_usa/dpl.rf

SEE ALSO

WindowInit(3W).

NAME

InputForm – get input from programmatic menus

SYNOPSIS

```
#include <dpl.h>
int InputForm( pForm )
form_t *pForm;
```

DESCRIPTION

InputForm is used to input values after programmatic menu construction.

This call, along with the *InitForm*, *ResetForm*, *DisplayForm*, *AddItem*, and *CloseForm* calls, substitutes for the *PopupForm* call that is used in applications that build their own menu items.

EXAMPLES

```
#include <dpl.h>

main()
{
        form_t *MyForm;
        char FieldValue[80];
        int Finished;
        int rc;

        InitForms();
        Init_MyFile();
        WindowInit(0);
        GetFormPtr( "MyForm", &MyForm );

        Finished = 0;
        OpenForm( MyForm );

        while ( ! Finished ) {
                rc = InputForm( MyForm );
                if ( rc == ERR_BADKEY ) {
                  ErrorForm("You typed an illegal key.",0);
                  continue;
                  }
                if ( rc == ERR_IO && errno == EINTR )
                        continue;
                Finished = 1;
                }

        if ( rc != ERR_CANCELED ) {
          GetFieldValue( MyForm, "MyField", FieldValue );
          NoteForm( "Your answer was $1 ", FieldValue, 0 );
          }

        CloseForm( MyForm );
        WindowExit( 0 );
}
```

Because of the mechanism by which CTIX signals work, the *InputForm* call may return before input is complete. This would happen if a signal came in during the time that the user is entering input. Here is one way that this is handled:

```
        .
        .
        .
while( InputForm( pForm ) == ERR_IO && errno == EINTR )
        ;
        .
        .
        .
```

NAME

KeyForm – feed keystrokes to a form

SYNOPSIS

```
#include <dpl.h>
int KeyForm( pForm, ch )
form_t *pForm;
short ch;
```

DESCRIPTION

In a mode where the programmer wants to use the *OpenForm*, *InputForm*, and *CloseForm* calls in place of the *PopupForm* call, this call is used to feed keystrokes to the form system. It is used instead of the *InputForm* call. Typically, this call is used to perform type checking on each keystroke, prior to the user filling in an entire field.

EXAMPLES

```
form MyForm
        field edit "          ";



#include <dpl.h>
main()
{
        short ID;
          .
          .
          .
        OpenForm( pForm );
        GetWindowId( pForm, &ID );
        DisplayForm( pForm );
        while( ( ch = wgetc( ID ) ) != LastKey ) {
                if ( ch != BadKey ) {
                        KeyForm( pForm, ch );
                        DisplayForm( pForm );
                } else
                        NoteForm( "Bad key.  Try again", 0 );
        }
        GetFieldValue( pForm, "MyField", FieldValue );
        CloseForm( pForm );
          .
          .
          .
}
```

NAME

MessageOn – leave a message in a window

SYNOPSIS

int MessageOn(format [, arg] ..., 0)
char *format;

int MessageOff( )
char *format

DESCRIPTION

MessageOn creates a window, and displays a message in it, but does not wait for the user to press a key before returning control to the program. The message is left on the screen until the program calls *MessageOff*. The *format* value is a character string with an embedded $1, $2, $3, etc. into which the *arg* values are embedded. The function value returned is an error value.

EXAMPLES

```
char *currentfile;
  .
  .
  .
/* let user know what we are doing */
MessageOn( "Reading in file $1", currentfile, 0 );
  .
  .
  .
MessageOff();
  .
  .
  .
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

NoteForm(3D), ErrorForm(3D).

NAME

NoteForm – display a message in a window

SYNOPSIS

```
int NoteForm( format [, arg ] ..., 0 )
char *format;
char *arg;
```

DESCRIPTION

NoteForm creates a "popup window" and displays a message in it, waiting for the user to press a key before restoring the screen to its state prior to the message. The *format* value is a character string with an embedded $1, $2, $3, etc. into which the *arg* values are embedded. Like *PopupForm*, the error return could be the value ERR_CANCELED, if the Cancel key is used to exit the form.

EXAMPLES

```
NoteForm( "$1, $2", "Hello", "world", 0 );
```

NAME

OpenForm – prepare a form to be displayed

SYNOPSIS

#include <dpl.h>
int OpenForm(pForm)
form_t *pForm;

DESCRIPTION

OpenForm is used to prepare a form to be displayed. *OpenForm*, *InputForm*, and *CloseForm* combine together to make up the functionality of *PopupForm*, but afford the user the capability of checking user input before the form is cleared from the screen.

EXAMPLES

```
#include <dpl.h>
main()
{
        form_t *MyForm;
        char FieldValue[80];
        int err;
        int Finished;

        InitForms();
        Init_MyFile();
        WindowInit(0);

        GetFormPtr( "MyForm", &MyForm );

        Finished = FALSE;
        OpenForm( MyForm );

        do {
          InputForm( MyForm );
          GetFieldValue( MyForm, "MyField", FieldValue );
          if ( strcmp( FieldValue, "The Right Answer" ) )
                NoteForm( "Bad answer. Try again.", 0 );
          else
                Finished = TRUE;

        } while( Finished != TRUE );

        CloseForm( MyForm );

        WindowExit(0);
}
```

NAME

OpenFormFile – open resource file with forms definition

SYNOPSIS

int OpenFormFile(FileName)
char *FileName;

DESCRIPTION

OpenFormFile is used for forms applications that do not use *rcc*, the forms compiler, but instead access their forms from a file at runtime. *FileName* is the name of the resource file. More than one *OpenFormFile* call per program is legal. Forms read in with *OpenFormFile* remain for the duration of the process, i.e. there is no need for a "CloseFormFile" call.

EXAMPLES

```
#include <dpl.h>

main()
{       int err;
        form_t *MyForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms initialization error \n");
                exit(1);
        }
        WindowInit(0);
        if (( err = OpenFormFile( "MyForms.rf" ) != 0 ) {
                ErrorForm( "Problem with the forms file ", 0);
                WindowExit(1);
        }

        GetFormPtr( "MyForm", &MyForm );
        if (( err = PopupForm( MyForm, 0 ) ) != 0 ) {
                ErrorForm( "Forms call error ", 0 );
                WindowExit(1);
        }
        WindowExit(0);
}
```

NAME

PopupForm – display a form and get user input

SYNOPSIS

```
#include <dpl.h>
int PopupForm( pForm, [ arglist..., ] 0 )
form_t *pForm;
char *arg;
```

DESCRIPTION

PopupForm displays a form, gets user input, and restores the screen to its prior state when the user input is complete. The user input is stored internally and is accessible by means of the *GetFieldValue* call and other calls. The function return value is either zero or an error code. Note that if the Cancel key is used to exit the form, the error return is the value ERR_CANCELED. The *arglist* is a list of pointers to strings to be used as parameters to the form. This variable list must be terminated by a zero.

PopupForm is a high-level aggregate of four other form calls: *SetFormArgs*, *OpenForm*, *InputForm*, and *CloseForm*. Applications requiring finer control over their forms should use the lower level calls.

EXAMPLES

```
form MyForm "Pick A File" (2,2)
        field FileName menu (2,2)-(4,25)
                "$1",
                "$2",
                "$3";

#include <dpl.h>
main()
{       int err;
        form_t *MyForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        Init_MyFile();
        WindowInit(0);
        if (( err = PopupForm( MyForm, "Item1", "Item2",
                                        "Item3", 0))!=0){
                if ( err != ERR_CANCELED )
                        ErrorForm( "Forms call error ", 0 );
                WindowExit(1);
        }
          .
          .
          .
        WindowExit(0);
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**. The error code ERR_CANCELED is returned if the user pressed the Cancel key to exit the form.

SEE ALSO

SetFormArgs(3D), OpenForm(3D), InputForm(3D), and CloseForm(3D).

NAME

RefreshField – redisplay a field of a form

SYNOPSIS

```
#include <dpl.h>
int RefreshField( pform, FieldName )
form_t *pform;
char *FieldName;
```

DESCRIPTION

RefreshField is used to recalculate the items in a field when some action has caused that field to be inaccurate. If the field was initialized by means of a shell script, that shell script is run again.

NAME

ResetForm – prepare form for programmatic menu construction

SYNOPSIS

#include <dpl.h>
int ResetForm(pForm)
form_t *pForm;

DESCRIPTION

ResetForm is used to prepare the form for programmatic menu construction, by means of the *AddItem* call and other calls. It clears the existing field values from the form. It should be called in every case that menu items will be added to the menu.

This call, along with the *InitForm*, *DisplayForm*, *InputForm*, *AddItem*, and *CloseForm* calls, substitutes for the *PopupForm* call that is used in applications that build their own menu items.

EXAMPLES

```
form mainform "Demo menu" (2,2)
        field mainmenu (2,2)
                ;


#include <dpl.h>
main()
{
        form_t *pmainform;
        char    textbuffer[80];
        int     i;

        InitForms(0);
        WindowInit(0);

        OpenFormFile("MyFile.rf");
        GetFormPtr( "mainform", &pmainform );

        /* throw away previous contents */
        ResetForm( pmainform );

        for( i = 1; i <= 4; i++ ) {
          sprintf( textbuffer, "This is item #%d", i );
          AddItem( pmainform, "mainmenu", textbuffer,0,0);
          }
        InitForm( pmainform );   /* init new form */
        InputForm( pmainform );  /* get user selection */

        GetFieldValue( pmainform, "mainmenu", textbuffer );

        /* erase form from screen */
        CloseForm( pmainform );
        WindowExit(0);
}
```

NAME

SetEditDefault – set the default edit field value

SYNOPSIS

#include <dpl.h>
int SetEditDefault(pForm, FieldName, DefaultValue)
form_t *pForm;
char *FieldName;
char *DefaultValue;

DESCRIPTION

SetEditDefault sets the default value for an edit field. *FieldName* is the name of the edit field. *DefaultValue* is the character value to which the field gets initialized.

Since the *PopupForm* call re-establishes the default value when it is called, it is necessary to use the lower level *ResetForm*, *InitForm*, *InputForm*, and *CloseForm* calls with *SetEditDefault*.

EXAMPLES

```
form MyForm "Enter the Name of Your State" (2,2)
        field State edit (2,2)-(20,3)
                "";


#include <dpl.h>
main()
{       int err;
        form_t *MyForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        init_MyFile();
        WindowInit(0);
        GetFormPtr( "MyForm", &MyForm );
        ResetForm( MyForm );
        if (( err = SetEditDefault( MyForm, "State", "Iowa" ) !=0){
                ErrorForm( "SetEditDefault problem ", 0 );
                WindowExit(1);
        }
        InitForm( MyForm );
        InputForm( MyForm );
        CloseForm( MyForm );
          .
          .
          .
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

SetMenuDefault(3D), SetListDefault(3D).

NAME

SetFormArgs – set up arguments to a form

SYNOPSIS

```
#include <dpl.h>
int SetFormArgs( pForm, [arg1, arg2, ... ], 0 )
form_t *pForm;
char *arg;

int SetFormArgv( pForm, argv );
form_t *pForm;
char *argv[n];
```

DESCRIPTION

In a mode where the programmer wants to use the *OpenForm*, *InputForm*, and *CloseForm* calls in place of the *PopupForm* call, either of these calls is used to set up the arguments that *PopupForm* passes to the form. *SetFormArgs* takes a list of strings, while *SetFormArgv* takes an **argv**-style array of pointers to strings. The maximum number of arguments is defined by the constant **MAXARGC** in **dpl.h**.

EXAMPLES

```
form mainform
        field menu
                "$1",
                "$2",
                "$3";

#include <dpl.h>
main()
{
        .
        .
        .
        SetFormArgs( pForm, "Item1", "Item2", "Item3", 0 );
        OpenForm( pForm );
        InputForm( pForm );
        CloseForm( pForm );
        .
        .
        .
}
```

The four calls in the "C" example above are equivalent to this call:

```
        PopupForm( pForm, "Item1", "Item2", "Item3", 0 );
```

NAME

SetFormPos – specify a form's size and position

SYNOPSIS

```
#include <dpl.h>

int SetFormPos( pform, begy, begx, height, width );
form_t *pform;
short begy;
short begx;
short height;
short width;
```

DESCRIPTION

SetFormPos may be used to move or resize a form on the screen.

Pform identifies the form, and the other parameters are values in the *wstat* struct. [See *WGetArgs*(3W).]

NAME

SetLevel – set capability level of forms session

SYNOPSIS

int SetLevel(NewLevel)
int NewLevel;

DESCRIPTION

SetLevel is used to change the **level** value that is used with the *level* item attribute in a resource file. The *NewLevel* must be a value between 0 and 7.

EXAMPLES

```
form mainform
        field menu
                "For Superusers"      [level=0],
                "For Important users" [level=1],
                "For all users"       [level=7];
                .
                .
                .

#include <dpl.h>

main()
{
        .
        .
        .
        SetLevel( 0 );
        .
        .
        .
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

GetLevel(3D).

NAME

SetListDefault – set the default list item(s)

SYNOPSIS

```
#include <dpl.h>
int SetListDefault(pForm, FieldName, ListValue)
form_t *pForm;
char *FieldName;
char *ListValue;
```

DESCRIPTION

SetListDefault selects a list item as a default. This list item is typically marked with an asterisk when the list is displayed.

FieldName is the name of the menu field whose menu value is to be defaulted. *ListValue* is the value of the item to be marked as default.

This routine can be called multiple times, to set multiple defaults.

Since the *PopupForm* call re-establishes the default value when it is called, it is necessary to use the lower level *ResetForm*, *InitForm*, *InputForm*, and *CloseForm* calls with *SetListDefault*.

EXAMPLES

```
form MyForm "Pick One Or More Item" (2,2)
        field MyMenu list [chcksel] (2,2)
                "Item1",
                "Item2",
                "Item3";


#include <dpl.h>
main()
{       int err;
        form_t *MyForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        Init_MyFile();
        WindowInit(0);
        GetFormPtr( "MyForm", &MyForm );
        ResetForm( MyForm );
        err = SetListDefault( MyForm, "MyMenu", "Item2" );
        if ( err != 0 ){
                ErrorForm( "SetListDefault problem ", 0 );
                WindowExit(1);
                }
        InitForm( MyForm );
        InputForm( MyForm );
        CloseForm( MyForm );
          .
          .
          .
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

SetMenuDefault(3D), SetEditDefault(3D).

NAME

SetMenuDefault – set the default menu item

SYNOPSIS

```
#include <dpl.h>
int SetMenuDefault( pForm, FieldName, MenuValue )
form_t *pForm;
char *FieldName;
char *MenuValue;
```

DESCRIPTION

SetMenuDefault selects a menu item as the default. When the menu is initially displayed the cursor will be on the item specified by this call.

FieldName is the name of the menu field whose menu value is to be defaulted. *MenuValue* is the value of the item to be marked as default.

Since the *PopupForm* call re-establishes the default value when it is called, it is necessary to use the lower level *ResetForm, InitForm, InputForm,* and *CloseForm* calls with *SetMenuDefault.*

EXAMPLES

```
form MyForm "Pick An Item" (2,2)
        field MyMenu menu [chcksel] (2,2)
                "Item1",
                "Item2",
                "Item3";


#include <dpl.h>
main()
{
        int err;
        form_t *MyForm;

        if (( err = InitForms() ) != 0 ) {
                printf( "Forms system error \n" );
                exit(1);
        }
        Init_MyFile();
        WindowInit(0);
        GetFormPtr( "MyForm", &MyForm );
        ResetForm( MyForm );
        err = SetMenuDefault( MyForm, "MyMenu", "Item2" );
        if ( err !=0 ){
                ErrorForm( "SetMenuDefault problem ", 0 );
                WindowExit(1);
        }
        InitForm( MyForm );
        InputForm( MyForm );
        CloseForm( MyForm );
          .
          .
          .
}
```

DIAGNOSTICS

This routine returns zero if successful and an error code otherwise. The error codes are defined in **dpl.h**.

SEE ALSO

SetListDefault(3D), SetEditDefault(3D).

NAME

fonts – CTAM font mapping files

SYNOPSIS

/usr/lib/ctam/fonts/*.ft

DESCRIPTION

CTAM employs a font description database in order to map the virtual font set available to an application to the actual fonts available on the terminal. If no fonts beyond ASCII are available in the terminal then no font description file is needed. However, if the terminal is capable of displaying different fonts, then these fonts need to be described with a font description file.

It is assumed that the terminal has one or more alternate fonts that may be selected and de-selected by use of multi-character sequences. These character sets are referred to as "alternate character sets." CTAM allows up to three different alternate character sets to be used to describe fonts. In order to correctly map CTAM's idea of what characters are going to appear on the screen it is necessary to establish a mapping between CTAM's virtual fonts and the terminal's real fonts. First, the sequences to switch between the terminal's alternate character sets must be specified. For example, if the terminal has a special graphics font that is selected by the sequence (1Bh, 65h) and de-selected by the sequence (1Bh, 66h) then the font file would contain the following:

```
smacs2=Ef, rmacs2=Eg,
```

The font mapping for each CTAM virtual character set is specified by a string containing sequences of three tuples. The first character in each tuple gives the position in the CTAM virtual character set by the equivalent ASCII character. The second character gives the position in the terminal's font of the desired physical character also by the equivalent ASCII character. The third character specifies the alternate character set number to be used to display the character or a tilde (~) to indicate that the high order bit should be set when displaying the character. Alternatively, if the terminal has a font that exactly matches a particular CTAM virtual font, then that font may be specified by the name of the virtual font, followed by an equals, followed by a single character representing the alternate character set that must be used to display that font.

Names of virtual fonts include: usascii, ukascii, decmulti, decgraph, ctgraph, ctline, user1, user2, and user3. Alternate character set 1 should be specified in the terminal's terminfo description file using the smacs and rmacs capabilities. Two additional alternate character sequences may be defined in the font map file using smacs2, rmacs2, smacs3, and rmacs3.

EXAMPLES

The Fortune terminal has an alternate character set containing many of the same symbols as the CTAM CT Graphics character set. In the following example, nine of these special characters are mapped from the Fortune

Systems Graphics Character Set onto the CT Graphics virtual font:

```
smacs=^N, rmacs=^O,
ctgraph=X81 >>1 <01 $41 &51 =11 |<1 ??1 ~v1,
```

The mapping string is made up of nine three-tuples each specifying a single character. White space in the mapping string is ignored. The first three-tuple states that the 'X' position of the CT Graphics Character Set (58 hex) is displayed by outputting an '8' (38 hex) when the terminal is in alternate character set 1.

FILES

/usr/lib/ctam/fonts/*.ft - Terminal font description database

SEE ALSO

terminfo(4).

For descriptions of the **decgraph** and **decmulti** character sets refer to the *VT-220 Programmer's Reference Manual EK-VT220-RM* or equivalent. For descriptions of **ctgraph** and **ctline** character sets refer to the *Convergent Programmable Terminal Programmer's Guide*.

NAME

kbmaps – CTAM keyboard mapping files

SYNOPSIS

/usr/lib/ctam/kbmaps/*.kb

DESCRIPTION

CTAM programs access files in the directory **/usr/lib/ctam/kbmaps** (or a directory named by the **KBMAP** environment variable) to determine information about a terminal's keyboard beyond what is described by *terminfo*(4). The information in the terminal's keyboard description file supersedes whatever information is specified in *terminfo*. The files in **/usr/lib/ctam/kbmaps** consist of lines of three fields each. The first field specifies the internal name of a key. A complete list of valid internal names is contained in **/usr/include/kcodes.h.** The second field specifies what the terminal sends when that key is pressed. The third field is optional and if present gives the keycap label for the key.

Key Semantics

The semantics of CTAM metakeys vary from one application to another. However, since the internal names of some metakeys do not accurately reflect their common usages, a list of basic keys and their meaning is presented here:

RollUp	Scroll down
RollDn	Scroll up
Next	Next page
Prev	Previous page
Forward	Right arrow or character right
Back	Left arrow or character left
Up	Up arrow or line up
Down	Down arrow or line down
Home	Beginning of page
s_Home	End of page
Beg	Beginning of document
End	End of document
Next	Shift right arrow, next word
Prev	Shift left arrow, previous word
s_Forward	Control right arrow, full scroll right
s_Back	Control left arrow, full scroll left
ClearLine	Erase field
DleteChar	Character delete
InputMode	Toggle insert/replace mode

EXAMPLES

The following example is from a keyboard mapping file for a Fortune terminal.

```
#
# Fortune keyboard description file
#
F1          ^Aa^M     F1
F2          ^Ab^M     F2
F3          ^Ac^M     F3
F4          ^Ad^M     F4
F5          ^Ae^M     F5
F6          ^Af^M     F6
F7          ^Ag^M     F7
F8          ^Ah^M     F8
F9          ^Ai^M     F9
F10         ^Ak^M     F10
Help        ^A@^M     Help
s_Page      ^As^M     PrevScrn
Beg         ^AS^M     s-PrevScrn
Page        ^Au^M     NextScrn
End         ^AU^M     s-NextScrn
Home        ^AX^M     s-Up
s_Home      ^AY^M     s-Down
Next        ^AZ^M     s-Right
Prev        ^AW^M     s-Left
Enter       ^Aq^M     Execute
InputMode   ^Ar^M     Insert
DleteChar   ^At^M     Delete
```

WARNINGS

It is important to avoid ambiguities in keyboard definitions. If one key sequence is a subset of another key sequence, the shorter of the two will always prevail. A system integrator adding support for a new terminal should watch out for this potential problem as CTAM does not check.

SEE ALSO

fonts(4W), terminfo(4).

NAME

escape – window escape codes

DESCRIPTION

CTAM windows emulate an extended ANSI X3.64 style terminal where special sequences of characters embedded in the output stream control certain aspects of the window. These aspects include character display attributes like reverse video and underlining as well as scrolling and erasing. Sequences of special characters written to the window via *wprintf*(3W), *wputc*(3W), and *wputs*(3W) are interpreted by CTAM along with normal text.

There are three broad categories of control sequences: C0 controls, C1 controls, and multiple character sequences. C0 control sequences are the familiar ASCII controls such as 0Dh (carriage return) and 0Ah (linefeed). C1 control sequences may be sent in two ways, as a single eight-bit value or as the ASCII escape code 1Bh followed by a second character. Multiple character sequences all begin with the C1 control called the Control Sequence Introducer. The CSI control code may expressed as the single eight bit value 9Bh, or as the two character sequence 1Bh 5Bh (Escape [). This type of control sequence is used for more complex operations.

C0 Controls

Name	Sequence	Description
NUL	00h	Null (ignored)
BEL	07h	Sound Bell
BS	08h	Backspace if col > 1
HT	09h	Horizontal tab
LF	0Ah	Linefeed; scroll up at bottom of scroll region.
VT	0Bh	Vertical tab; down one or scroll up at bottom of scroll region.
FF	0Ch	Form Feed; same as VT
CR	0Dh	Carriage Return; cursor moves to column 1
SO	0Eh	Shift out; selects G1 character set for GL
SI	0Fh	Shift in; selects G0 character set for GL

C1 Controls

Name	Sequence	Description
IND	84h or Esc D	Index (same as linefeed)
HTS	88h or Esc H	Horizontal tab set
RI	8Dh or Esc M	Reverse Index; scroll down in row 1
SS2	8Eh or Esc N	Single shift G2 into GL for the next character
SS3	8Fh or Esc O	Single shift G3 into GL for the next character
NEL	85h or Esc E	New Line; move to column 1 of next line
CSI	8Bh or Esc [	Control Sequence Introducer; see below

Name	Sequence	Description
SC	Esc 7	Save cursor position and cursor attributes
RC	Esc 8	Restore cursor position and attributes.
LS1R	Esc ~	Lock shift G1 into GR
LS2	Esc n	Lock shift G2 into GL
LS2R	Esc }	Lock shift G2 into GR
LS3	Esc o	Lock shift G3 into GL
LS3R	Esc ¦	Lock shift G3 into GR

Multiple Character Sequences

Name	Sequence	Description
CUP	CSI Ps1 ; Ps2 H	Move cursor to column Ps1, row Ps2
CUU	CSI Pn A	Move cursor up Pn lines
CUD	CSI Pn B	Move cursor down Pn lines
CUF	CSI Pn C	Move cursor forward Pn columns
CUB	CSI Pn D	Move cursor back Pn columns
CNL	CSI E	Move cursor to column 1 of next line
CPL	CSI F	Move cursor to column 1 of previous line
SU	CSI Pn S	Scroll up Pn lines
SD	CSI Pn T	Scroll down Pn lines
DCH	CSI Pn P	Delete Pn positions
ICH	CSI Pn @	Insert Pn positions
ECH	CSI Pn X	Erase (change to space) next Pn positions
DL	CSI Pn M	Delete Pn lines
IL	CSI Pn L	Insert Pn lines
EL0	CSI 0 K	Erase cursor to end of line
EL1	CSI 1 K	Erase beginning of line to cursor
EL2	CSI 2 K	Erase entire line
ED0	CSI 0 J	Erase cursor to end of display
ED1	CSI 1 J	Erase beginning of display to cursor
ED2	CSI 2 J	Erase entire display
SGR0	CSI 0 m	Set all attributes to normal
SGR1	CSI 1 m	Select bold
SGR2	CSI 2 m	Select dim
SGR4	CSI 4 m	Select underline
SGR7	CSI 7 m	Select reverse
SGR9	CSI 9 m	Select struck out
SGR21	CSI 21 m	Turn off bold
SGR22	CSI 22 m	Turn off dim
SGR24	CSI 24 m	Turn off underlining
SGR27	CSI 27 m	Turn off reverse
SGR29	CSI 29 m	Turn off struck out

Name	Sequence	Description
TBC0	CSI 0 g	Remove horizontal tab stop at current position
TBC3	CSI 3 g	Remove all horizontal tab stops
CSR	CSI Ps1;Ps2 r	Set scroll region
DSR	CSI n	Device status report
CTSLP0	CSI = 0;Ps2 @	Move to prompt line, column Ps2 (see CTSLN)
CTSLP1	CSI = 1;Ps2 @	Move to tag line, column Ps2 (see CTSLN)
CTSLP2	CSI = 2;Ps2 @	Move to SLK line, column Ps2 (see CTSLN)
CTSLP3	CSI = 3;Ps2 @	Move to command line, column Ps2 (see CTSLN)
CTSLN	CSI = Ps1 q	Set the number of active noise lines (Before any special line positions (CTSLP0-3) can be used, CTSLN must be used.)
CTVIS0	CSI = 0 C	Make cursor visible
CTVIS1	CSI = 1 C	Make cursor invisible
CTMF	CSI = Ps1;.. R	Map fonts Ps1... to G0...
CTSU	CSI = Ps1;Ps2;Pn S	Scroll lines Ps1 through Ps2 up Pn lines
CTSD	CSI = Ps1;Ps2;Pn T	Scroll lines Ps1 through Ps2 down Pn lines
CTWN	CSI = W	Write window number
CTDSR	CSI = b	Device status report
CTSGR	CSI =Ps1;Ps2m	Select Ps1 = on mask; Ps2 = off mask
CTSM2	CSI = 2 h	Clear and enable window label
CTSM7	CSI = 7 h	Save cursor (same as SC)
CTTRON	CSI = 3 h	Enable cursor tracking
CTTROFF	CSI = 3 l	Disable cursor tracking
CTRM2	CSI = 2 l	Disable window label
CTRM7	CSI = 7 l	Restore cursor (same as RC)
CTRESET	CSI = p	Reset window to initial modes
CTSD	CSI = 0 w	Disable scrolling
CTSE	CSI = 1 w	Enable scrolling
DECCOLM	CSI ? 3 h	Set window width to 132 columns
DECOM	CSI ? 6 h	Set origin (1,1) to be top of scroll region
DECAWM	CSI ? 7 h	Enable autowrap at column 80

Name	**Sequence**	**Description**
DECTCEM	CSI ? 25 h	Same as CTVIS0
DECRM3	CSI ? 3 l	Set window width to
DECRM6	CSI ? 6 l	Set origin to be top of screen
DECRM7	CSI ? 7 l	Disable autowrap
DECRM25	CSI ? 25 l	Same as CTVIS1
DECDSR	CSI ? n	Device status report
DSG0	Esc (*F*	
DSG1	Esc) *F*	
DSG2	Esc * *F*	
DSG3	Esc + *F*	

Designate character set G0, G1, G2, or G3 as font *F* where *F* is: 'A' for UK ASCII, 'B' for US ASCII, '0' for DEC special graphics, '<' for DEC multinational, '=' for PT special graphics, '2' for user font 1, '3' for user font 2, or '4' for user font 3.

FILES

/usr/include/ctam.h

Glossary

action routine. An action routine has syntax approximating that of a C subroutine call and is invoked upon the occurrence of a particular event.

attribute. An attribute keyword (or flag) causes a particular characteristic of a form or field to be turned on or off. If an attribute keyword is preceded with a tilde (~), that attribute is turned off. Each field attribute is either on or off with the default depending on the field type. Some attributes are useful only with some field types, while other attributes may be used with any field type. Attributes are specified with a comma separated list of keywords enclosed by square brackets.

CTAM. CTAM (the Convergent Terminal Access Method) is an applications development kit for CTIX systems. CTAM provides a device-independent ANSI X3.64 interface to terminals and a library of routines for creating, manipulating, and displaying to windows.

default. A default is a value that is selected for you by the system unless you specify a different value.

dialogue. A dialogue is a session in which an end user interacts with forms and menus to get services provided by one or more application programs.

DPL. The Dialogue Programming Language is part of the CTAM development package. DPL is a high level language that enables a developer to create applications that display forms and menus.

edit field. An edit field is an area in the display that can be modified with keyboard input. See Figure 2-1 for more examples of field types.

event. An event defines the action or actions performed when a selection associated with a form, field, or single menu item is complete. Braces {} surround the action(s) to be performed.

field. A field is a subset of a form; it is an area on the screen that varies in shape and size, depending on the number and format of the items inside. Depending on the type of field, items in a field can be selected (menu fields) or edited (edit fields), or if the field is for viewing only, neither selected nor edited (text fields). Some fields take up the whole screen, others take up enough space for only a few characters.

flag. See attribute.

font description file. For terminals with font capability beyond ASCII, files with the suffix **.ft** in the directory **/usr/lib/ctam/fonts** enable CTAM to map the virtual font set available to an application to the actual fonts available on a terminal. For more information, see **fonts**(4w) in Appendix A.

form. A form is a display containing one or more fields, much like a hardcopy questionaire form, which might contain a mixture of multiple choice (menu field), fill in the blank (edit field), and general information (text field) items.

internal value. The internal value is the octal code (defined in **/usr/include/kcodes.h**) used by CTAM routines to identify a particular keystroke sequence (see Table 3–1).

item. An item is a piece of text within a field. Normally, items appear one per line in a single column.

keyboard description file. Files with the suffix **.kb** in the directory **/usr/lib/ctam/kbmaps** contain information about a terminal's keyboard beyond what is described by **terminfo**(4). For more information, see **kbmaps**(4w) in Appendix A.

keyword. A DPL keyword is a predefined word that has a specific meaning in DPL programs. For a list of various keywords and the pages on which they appear, see "keywords," in the Index.

main menu. The main menu is the top of an application's menu hierarchy, providing access to the major functions within the program.

major number. The major number represents a class of devices, such as terminals or printers.

menu. A menu is a field containing a list of choices (items) from which you make a selection. A menu can take up an entire screen or

be part of a form. On certain forms, *pop–up* menus can be displayed by pressing a programmed sequence of keys.

metakey. Metakeys are keys function keys or action invoking keys, such as **Scroll Up** or **Delete**, that can be programmed to invoke various functions, depending on the current controlling process.

minor number. The minor number represents the specific device within a class of devices.

pop–up menu. A pop–up menu is an optional menu display that is used when there is not sufficient space in the current form to display a list of items. Pop–up menus are invoked from fields with this option by pressing a programmed key sequence.

resource file. A resource file is a file with the suffix **.rf** that contains menu and form information interpreted by **dplrun**.

special file. A special file is an entry in the **/dev** directory that is associated with a device driver; it is a way of accessing a device such as a terminal or window (virtual terminal).

text field. A text field contains items that are for viewing only; that is, the items can not be selected or edited. The prompt line is a good example of a text field. See Figure 2–1 for more examples of field types.

type verification string. Type verification strings are used to define allowable characters in an input string so that characters entered into an edit field can be evaluated as to their validity.

window terminal. A window terminal is a software construct presented to an application process by the window manager (**ctwm**).

Index

$TERMctam.ti
listing of, 3-15
/dev/ttynnn, 3-5
/dev/window, 3-5
/dev/wxt/wnnn, 3-5
/dev, 3-4
/etc/CTWMtermcap, 3-12
/etc/drvload, 3-5
/etc/lddrv/wxt.o, 3-5
/etc/master, 3-5
/etc/termcap, 3-12
/etc, 3-4
/usr/include/kcodes.h, 3-8, 4-24
/usr/lib/ctam/english_usa/ctwm.rf, 3-6
/usr/lib/ctam/english_usa/dpl.rf, 3-6
/usr/lib/ctam/fonts/$TERM.ft, 3-11
/usr/lib/ctam/kbmaps/$TERM.kb, 3-8
/usr/lib/libctam.a, 1-2
/usr/lib/libdpl.a, 1-2
/usr/lib/libxnls.a, 1-2, 6-1
/usr/lib/terminfo/?/$TERM, 3-6
/usr/lib/terminfo/?/$TERMctam, 3-6

A

action routines, 4-24
AdvanceField, 4-28
BackField, 4-28
CloseForm, 4-28
doexec, 4-24
ErrorForm, 4-26
GotoForm, 4-28
LabelForm, 4-28
LabelKey, 4-27
NoteForm, 4-25
PopupForm, 4-25
RefreshField, 4-27
SetCurrentField, 4-28
SetPrompt, 4-26
SetRefreshRate, 4-26
SetSelect, 4-28
active process, 4-6
ANSI x3.64, 1-2, 3-3
arrow keys, 2-3

B

Backspace, 2-3
Back, 2-3
BASIC, 7-3

C

C routines
GetFieldValue, 5-3
GetFormPtr, 5-3
GetLevel, 4-19
InitForms, 5-3
Init_MyFile, 5-4
NoteFrom, 5-3
OpenFormFile, 5-3
PopupForm, 5-3
SetLevel, 4-19
WindowExit, 5-3
WindowInit, 5-3
Cancel, 2-4
COBOL, 7-1
control codes, 1-4, 2-4
control flow, 4-29
control flow operators, 4-31
Control, 1-4

conventions used, 1-4
CTAM
  components, 1-1
  defined, 1-1
  window manager, 1-1, 3-1, 3-4, 4-8
ctwm(1W), 1-1, 3-1
curses(3), 3-3 to 3-4
cursor, 2-3

D

default values, 2-4
Delete, 2-3
dialogue, 2-1, 4-2
Down, 2-3
DPL, 2-1, 4-1
dplrun(1), 1-1

E

Enter, 2-4
events, 4-20
  control flow, 4-29
  onact, 4-20
  onbadkey, 4-20
  oncancel, 4-20
  onclose, 4-20
  onhelp, 4-20
  oninit, 4-20
  onkey, 4-21, 4-23
  onselect, 4-21
  onvalid, 4-21
  scoping of, 4-22
  traps, 5-5

F

fields, 2-2, 4-5
  attributes, 4-9
    blank, 4-14
    boxed, 4-10
    chcksel, 4-12
    chck, 4-12
    dash, 4-12
    hbar, 4-10
    highsel, 4-12
    high, 4-12
    off, 4-10
    pulldown, 4-12
    save, 4-10, 4-13
    tail, 4-17
    vbar, 4-10
    writein, 4-12
  coordinates, 4-9
  edit, 4-5, 4-13
    initializing, 4-13
    type checking, 4-14
  editing, 2-2
  list, 2-4, 4-5, 4-12
  menu, 4-5, 4-10
  multiple column, 4-11
  scrollable, 2-2, 4-17
  selecting, 2-2
  specifying, 4-8
  text, 4-5, 4-17
  turning off attributes, 4-9
  types, 4-9
  view only, 2-2
Figures
  CTAM Window Manager, 1-2
  Hierarchy of DPL Entities Under CTWM, 4-6
  Sample Form #1, 2-2
  Sample Form #2, 2-3
  Sample Form #3, 4-3
  Sample Form #4, 4-4
  Sample Form #5, 4-5
  The User/Kernel Interface Under CTAM, 3-2
  Using **terminfo** Description Files, 3-7
Finish, 2-4
font description file, 3-11
forms, 2-1 to 2-2, 4-2, 4-5
  compiler, 1-2
  coordinates, 4-7
  flags, 4-8
    fullwidth, 4-8
    new, 4-8
    popup, 4-8
    resize, 4-8
    son, 4-8
  how to specify, 4-7
  interpreter, 1-1
  labels, 4-7
  library, 1-2
  moving between, 2-3
  moving within, 2-3
FORTRAN, 7-4
Forward, 2-3

H

Help, 2-5

I

internal value, 3-2
items, 2-2, 4-5
  attributes, 4-18
    blank, 4-19
    default, 4-19
    high, 4-19
    level, 4-18
    root, 4-18
  display value, 4-20
  specifying, 4-17
  user value, 4-20

K

keyboard description file, 3-8
keypad modes, 3-2
keys
  arrow, 2-3
  Backspace, 2-3
  Back, 2-3
  Cancel, 2-4
  Control, 1-4
  Delete, 2-3
  Down, 2-3
  Enter, 2-4
  Finish, 2-4
  Forward, 2-3
  Help, 2-5
  Mark, 2-4
  Return, 2-3
  Tab, 2-3
  Up, 2-3
  virtual, 1-4, 3-2, 3-8
keywords, 4-2
  edit, 4-9
  else, 4-29
  field, 4-8
  form, 4-7
  fullwidth, 4-8
  if, 4-29
  list, 4-9
  menu, 4-9
  new, 4-8
  popup, 4-8
  resize, 4-8
  return, 4-29
  son, 4-8
  text, 4-9
  while, 4-29

L

lddrv(1), 3-5
libctam.a, 3-3

M

major number, 3-5
Mark, 2-4
menus, 2-1 to 2-2, 4-2
  moving between, 2-3
  moving within, 2-3
minor number, 3-5

O

operators, 4-31

P

physical terminals, 3-1

R

rcc(1), 1-2, 5-3
related documentation, 1-5
reserved variable names, 4-38
resource file, 4-2, 5-1
Return, 2-3

S

scroll bar, 2-2
special files, 3-4
stdio(3S), 3-4

T

Tab, 2-3
terminal specific, 3-7
terminals
  physical, 3-1
  supported, 3-12
  window, 3-1
terminfo(4), 3-6

tic(1), 3-15
tset(1), 3-12
type verification string, 4-14

U

Up, 2-3

V

variables, 4-33
  environment
    $HOME, 4-39
    $LANG, 3-6
    $LOGNAME, 4-39
    $SHELL, 4-39
    $TERMCAP, 3-12
    $TERM, 3-3, 3-6, 3-12
  global, 4-33, 4-37
  legal operators, 4-35
  reserved, 4-38
  special, 4-38
    $Cancl, 4-38
    $Cmd, 4-38
    $Enter, 4-38
    $ERROR, 4-38
    $Exit, 4-38
    $Help, 4-38
    $KEY, 4-38
    $LEVEL, 4-40
    $Tab, 4-40
vertical scroll bar, 2-2
virtual keys, 3-2, 3-8

W

wgetc(3W), 3-2
window devices, 3-1, 3-5
window driver, 3-1
window manager, 1-1, 3-1, 3-4, 4-8
window terminals, 3-1
windowing library, 1-2
wxt, 3-1, 3-4